AFRL-IF-RS-TR-2002-166
Final Technical Report
July 2002

SECURITY ENGINEERING FOR HIGH ASSURANCE, POLICY-BASED APPLICATIONS

Odyssey Research Associates

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E298

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-166 has been reviewed and is approved for publication.

APPROVED: JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Jul 02	Final Sep 96 - Sep 99

4. TITLE AND SUBTITLE
SECURITY ENGINEERING FOR HIGH ASSURANCE, POLICY-BASED APPLICATIONS

5. FUNDING NUMBERS
C - F30602-96-C-0303
PE - 62301E
PR - E017
TA - 01
WU - 08

6. AUTHOR(S)
David Rosenthal, Francis Fung, Stephen Garland, Andrew Myers and David Evans

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Odyssey Research Associates
Cornell Business & Technology Park
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFGB
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2002-166

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: John C. Faust, IFGB, 315-330-4544

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This report describes a research effort to define methods of analysis, components, and tools for handling information in an environment with complex trust and security relationships. The effort consists of two related tasks. The first task is to provide proper access control for data that is shared by multiple organizations in a networked environment. In particular, we describe an access control mechanism that factors security administration among different administrative entities. The access control language is object-oriented and facilitates the construction of default policies for newly created objects. The second task is to provide methods for describing and achieving the proper behavior of programs that may be executed in accessing shared data. In particular, we have designed three ways in which to express aspects of proper program behavior, developed program checking strategies for all three, and produced languages and checking tools for two of them. 1) PolyJ is a tool-supported extension to Java that provides improved compile-time assurance for the correctness of Java programs. 2)IFlow is a language and static checking strategy for describing and controlling information flow. 3) Naccio is a tool-supported code-transformation system for ensuring that executable mobile code adheres to user-defined security policies.

14. SUBJECT TERMS
Information Assurance, Access Control Program Behavior, Java, Program Correctness, Information Flow, Safety Property Enforcement, Mobile Code Security, Security Policy Specification Languages

15. NUMBER OF PAGES
96

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Abstract

This report describes a research effort of ORA and MIT to define methods of analysis, components, and tools for handling information in an environment with complex trust and security relationships. The effort consists of two related tasks.

The first task is to provide proper access control for data that is shared by multiple organizations in a networked environment. In particular, we describe an access control mechanism that factors security administration among different administrative entities. The access control language is object-oriented and facilitates the construction of default policies for newly created objects. We provide a description of the specification language and a high-level description of a demonstration implementation for web-based information sharing.

The second task is to provide methods for describing and achieving the proper behavior of programs that may be executed in accessing shared data. In particular, we have designed three ways in which to express aspects of proper program behavior, developed program checking strategies for all three, and produced languages and checking tools for two of them. (1) PolyJ is a tool-supported extension to Java that provides improved compile-time assurance for the correctness of Java programs. In particular, PolyJ uses static checking of parameterized data types to detect errors that would otherwise cause Java programs to fail during execution. (2) IFlow is a language and static checking strategy for describing and controlling information flow. It complements access control mechanisms by allowing different administrative entities to limit the dissemination of information to which they have granted access. (3) Naccio is a tool-supported code-transformation system for ensuring that executable mobile code adheres to user-defined security policies.

Table of Contents

List of Figures

Summary

This report describes the final results of work done by ORA and MIT to define methods of analysis, components, and tools for handling information in an environment with complex trust and security relationships. The effort consists of two related tasks, access control and program behavior.

Access Control

In order for access control methods to be effective, the access constraints on a resource need to reflect the intended policy. Unfortunately, typical users do not usually take the time to set up access constraints. This problem can be mitigated to some extent by the use of mandatory system level policies and default policies on newly created resources. However, the flexibility of such policies is usually limited, and so the constraints imposed are often not a good fit with user needs.

We have developed a language in which users can naturally specify default access constraints with great flexibility and precision. The highlights of this language are:

- It is object oriented, facilitating a natural representation of entities such as users and resources. The language is based on UML/OCL (Unified Modeling Language/Object Constraint Language), which is achieving wide acceptance as a standard for object modeling.

- The language includes constructs to express both *attributes* of users and resources, and *relationships* between them. Relationships can provide a generalization of the familiar *group* concept in access control. UML/OCL provides the required expressive power and is particularly appropriate for expressing access constraints involving relationships between entities, such as:

  - an organizational hierarchy, or
  - a connection between current projects and members of an organization.

- Relationships and (dynamic) attributes can be defined by separate administrative authorities by means of certificates. This permits an appropriate partition of administrative responsibilities.

The language for describing access rules involves more than just expressing the terms to be evaluated. It is supplemented with declarations of the classes, attributes, and relationships that appear in the expressions, in order for users to properly interpret the meanings of the rules.

When multiple users or organizations share resources, multiple policies may apply. For example, one organization may control a directory of a file system hierarchy, while an independent organization controls a subdirectory; each organization needs some control

over access to files of the subdirectory. The language contains constructs to control how the policies of different authorities are combined.

In order to demonstrate the usefulness of the approach, we have created a demonstration system that implements access control for a web-based information sharing system. We have used the demonstration system to study issues of expressiveness (does the language provide the expressiveness that users require in complex environments) and usability (is the language simple enough for ordinary users).

Program Behavior

Our work on assessing program behavior has three components:

- Languages in which to express security policies or desired behaviors.
- Static analysis tools (similar to the LCLint tool [Evans96] developed earlier at MIT) that use "light weight formal methods" to check whether code obeys the security policies and exhibits the desired behaviors.
- Run-time support for enforcing the security policies.

We have used this three-pronged approach to address issues concerning the reliability and safety of mobile code, as well as the flow of information from modules that have acquired access to that information via access-control mechanisms.

We have developed several means for increasing the reliability of applications written in Java. One enables better compile-time detection of programming errors. It provides an extended version of Java, called PolyJ, for use by programmers to describe the intended use of parameterized types and for use by static checking tools to ensure that these types are used as intended. The second provides support for improving the accuracy and utility of the documentation. Like the Sun javadoc utility, it extracts documentation from comments embedded in Java source code; unlike javadoc, it checks that the comments bear some relation to the source code.

In order to ensure the safety of mobile code, we have designed and developed tools that transform an arbitrary program into a trusted program that satisfies user-specified safety properties. Typical properties might restrict the files that mobile code can read or write, or limit how mobile code uses system resources. Supporting tools can be used to enforce these properties for downloaded Java classes or for Windows executables.

We have developed a new model, IFlow, of information flow that supports fine-grained, user controlled, dynamically changing releasability and downgrading constraints on data. This model encompasses multiple trust domains, an explicit fine-grained downgrading policy, and low enforcement overhead, most of which is incurred by static checking at compile time rather than by expensive run-time monitoring of program execution.

PART I: Introduction

1 Introduction

1.1 Scope and Objective

This report describes work performed at ORA and MIT to define methods of analysis, components, and tools for handling information in an environment with complex trust and security relationships. The scope of the effort is to:

- Provide proper protection for data that is shared by multiple organizations in a networked environment.

- Assess and achieve the proper behavior of programs that may be executed in the course of accessing the data.

For access control, our objective is to provide a means to precisely establish and enforce a security policy, in a way that minimizes the burden on the end-user. We present a method to achieve this objective by providing an expressive access control language that can be used to specify the initial policy on newly created objects.

For proper program behavior, our objectives are to provide improved support for assuring the correctness of Java programs, to enforce appropriate information flow (including appropriate downgrading), and to ensure that mobile code respects user-defined safety policies.

1.2 Report Description

The remainder of this report consists of the following parts:

- Part II describes the access control method with particular attention to the access control language.
- Part III describes language extensions and tools used to provide better compile-time checking that Java programs behave as expected.
- Part IV describes the mechanisms used to enforce the adherence of mobile code to user-defined safety properties.
- Part V describes the information flow model and the methods that can be used to check statically that code respects information flow policies.
- Part VI summarizes our results.

PART II: Access Control

2 Access Control Approach

2.1 Concept

The intent of this part of the project is to provide security support for information sharing between organizations. Different organizations (and suborganizations) may want to impose different kinds of restrictions on how data can be accessed. This is usually accomplished by some form of configurable access control on data. In the situation we are examining, we would like to support fine-grained policies. There have been many approaches to support fine-grained access control for information sharing. One of the most common approaches involves the use of some kind of access control list. However, a problem with this approach is that users typically do not bother to use it. For fine-grained policy methods to work effectively, there needs to be a mechanism to set the correct policies with minimal end user intervention.

If the information sharing application were totally within a single organization and the policy principles were relatively stable, one could hard-wire the access control support into an application and administer it centrally. Here we examine a situation in which different organizations may want to control and configure their policies. Hence, we want a more flexible way to define a policy than using built-in programs that enforce the policy. We also want to allow organizations to have more direct control over assigning the user characteristics upon which access decisions will be based.

To achieve the needed flexibility, we have developed a policy language for access rules that is sufficiently general to utilize user and object characteristics. This general idea is similar to other current approaches, such as Adage [ZBCS97]. In the approach presented here, the developers of the information sharing application specify a fuller object model (including relationships as well as attributes), and the application users can utilize a more expressive access constraint language than in Adage. Access rules are specified using the vocabulary of the object model. This provides an expressive and adaptable way to describe constraints. For example, an access rule can require that a user or file has a particular kind of security level attribute or that a user has a relationship indicating that they work for another user.

Relationships can be used to support a generalization of groups, called "parameterized groups". For instance, to represent the parameterized group "reportsto$_A$", which describes collections of users to whom a certain user A reports, we can use the relationship "A reportsto B". This construction permits users to compactly state rules that apply uniformly to many different objects and users. Other examples of parameterized groups are:

- users working on a certain project
- users permitted to handle a certain kind of data

In order to minimize the user burden in setting access controls, we provide a means of automatically setting default access policies on newly created objects. Since we can define access rules in terms of user relationships and attributes, we have greater capability to write rules that will form the correct policy on many different objects. We recognize that even if most users may not bother to properly set access controls, some users may want to alter a part of the policy that gets set by default. We provide two different ways of handling this. Users can build rules using the general access control language. A simpler method, a generalization of access control lists, is also available.

Another goal is to minimize the amount of administrative interaction between the cooperating organizations and the centralized information server. Organizations may use relationships to specify subsets of their users without having to directly interact with an information server administrator. For example, the structure of users within a participating organization can remain that organization's responsibility. This potentially remote specification of users and their relationships can be implemented in a secure way by using certificates. Certificates for user characteristics can be issued by certificate authorities (CA's) that are not directly associated with the authority managing the shared information.

A potential disadvantage of our approach is that the generality provided by the access language may make it harder to optimize access rule evaluation. However, this method is being proposed for information sharing between organizations, and so the number of access requests should be small enough that small delays in rule evaluation will not have a significant impact.

Another concern is that the generality of the rules may make them too difficult for users to understand. We anticipate that implementers of this access control mechanism may want to build specialized front-end languages to our language that are more limited and focused. We describe an example of a front-end language based on the concept of an access control list.

2.2 Access Control Policy

When the resources being managed have a natural hierarchy, such as a file system, it is possible to impose a layered access policy. A top-level policy can specify the part of the access control policy (a "base policy") that applies to every resource object. In this way, all users of the application are subject to a set of rules that are enforced on all resources. At another layer down, organizations can specify part of the policy on those objects that they "control". Lower layers can impose further restrictions depending on how that particular part should be controlled.

2.3 Example

To motivate the usefulness of our access control method we provide a simple example with part of a specification for it. Later, in Section 7, we discuss a more complete example with a fuller specification, including declarations for the classes of objects involved in the object model.

Suppose the Air Force Research Laboratory (AFRL) wants to set up a directory so that researchers at several different sites can share information. Each research site will have its own subdirectory, and should have some control on who can access information in that subdirectory. This includes defining which members of their own organization can use their subdirectory.

An informal explanation of some of the rules is as follows:

AFRL maintains a directory called `shared_project`. Each company can create a subdirectory of `shared_project`, e.g., `CompanyA`. A PI (Principal Investigator) for company A can create a subdirectory of `CompanyA`, say `Project_PI_A`. The PI has full rights to this project directory. The company also has the ability to establish that someone works for the PI (via a certificate), say employee `x`. With these credentials, user `x` has the right to create a subdirectory of the PI directory.

A rule such as:

```
(request.requestor=request.file.owner) or (request.operation=READ)
```

when associated with a file, will permit any operation by the file owner and read access by anyone else.

The rules governing access to a given file are not necessarily controlled by just one individual; they are a composite of the policies of the different authorities. In this example, AFRL can form a policy that all files and subdirectories will inherit. A company can impose additional constraints on its files, and so forth.

2.4 Target Objects

The concepts developed in this effort can apply to a variety of information sharing systems. However, the focus of this effort is on applications that share information using files and directories. In normal security terminology, these would be referred to as the objects of the system. However, we will be using the object-oriented modeling community's terminology. In such a context, the word "object" is more general, and could even refer to parts of the system that maintain information about subjects (e.g., users). To avoid confusion, we follow the Adage convention and use the words *target*, or *target object*, to refer to the application objects that are access controlled.

2.5 Demonstration System

We have implemented these concepts in a demonstration Web-based information sharing system.

2.5.1 The handling of a request

The system distinguishes between two different kinds of requests: requests to perform some operation such as a read or write on some target object, and requests to view or update access control information.

To perform an operation on a target object, a user first generates a request. (In our demonstration prototype there is a simple web-based GUI for making requests and receiving results.) The request is sent to the "information server", i.e., the part of the information sharing application that maintains and controls access to the data. The information server then performs the following steps:

- it evaluates the information in the request to see if it is "trusted"
- it checks the access rules using information supplied by the request and additional information maintained by the application
- if the access checks are passed, the server invokes the requested operation on the object (for example, a file may be updated)
- it then sends results of the request back to the user

The second kind of request involves updating access information. When a target object is first created, a security policy is initially assigned to that object. This includes the policies inherited from higher parts of the hierarchy (as described below) and a copy of the default user policy of the object creator. An object creator can update its part of the policy.

2.5.2 Policies

One of the features of this kind of information sharing application is that multiple parties may want to influence the allowable access to a given piece of information. First, all of the users of the application may agree to follow a common set of rules (e.g., no outsiders can directly gain access). Additionally, users may want additional constraints on "their" information. Furthermore, a user might control a directory and thus may wish to impose restrictions on files within that directory, even if those files are under another user's control.

To accommodate this, the application supports combining multiple policies from all of the parties that have some controlling interest on a target object. This policy structure can be thought of as being associated with the directory structure of the target objects. Inheritable policies apply to a target object and all lower level objects. Local policies apply only to that particular target object. The highest level policies (associated with the top of the directory hierarchy) are set based on how the resources at that site must be used for the proposed information sharing activity. The manager(s) of the application can allow other parties to impose further restrictions on subdirectories. This policy structure is designed to let one party have control over a given parent directory (by which we mean the ability to manipulate security-relevant attributes of the directory such as local policy), while another party has control of a subdirectory. For instance, a controller of a subdirectory may be able to restrict the access rights of the parent directory's controller.

In the "normal mode" of operation, user A can grant user B the rights to create and use a subdirectory in A's directory. The "normal" use does not necessarily give A any access rights to a subdirectory created by B. That is, B's rights on the subdirectory may exceed the rights of A on that subdirectory. The current design anticipates the use of "special

permissions" for a controlling party to take control of a subtarget in special circumstances (see Section 5.6.4).

2.5.3 Certificates

Information about individuals, their characteristics, and relationships may be issued by authorities not directly connected with the management of the information server. Indeed, such information may be issued from different organizations at possibly different locations. The information server needs a way to establish the trustworthiness of information submitted to it. This is handled by the use of certificates.

Certificates consist of data that has been digitally signed by an authority, i.e., a certificate authority (CA). The information server can check the validity of the signatures on the certificates.

For the demonstration, we provide some certificate support for identity authentication, but only simulated support for conveying attributes and relationships via certificates.

2.6 Adage

Adage is an access control system developed by the Open Group Research Institute, as a DARPA effort, that has some similarities to the methods proposed here. It places considerably more emphasis on role based access control (RBAC) and has less emphasis on the expressive power of object relationships and attributes in access control rules.

In several places in this document, references are made to Adage for comparison purposes. However, knowledge of Adage is not necessary to follow this report.

2.7 What Follows

We divide the description of the policy language into three main sections. The primary part of a security policy will be rules governing access. Section 3 describes the expression language used to form the rules to be evaluated. Those expressions refer to particular classes, objects, and relationships. Syntax for defining those elements is given in Section 4. Section 5 describes how the access rules can be grouped together as policies.

Section 6 describes the classes for an information sharing system, i.e., the predefined classes and objects. Section 7 presents an information sharing example illustrating the approach. In Section 8, we provide a short description of an information sharing demonstration that was built to demonstrate these concepts.

3 Expression Language

Part of making an access control decision involves evaluating boolean expressions that involve the relationships and attributes of objects (including users). These expressions form the content of the rules and evaluate to either true or false depending upon whether access should be allowed. In this section, we describe the part of the policy language for forming such expressions.

The expression language is essentially OCL [OCL], which is a language for forming constraints on objects in the language UML. We chose it because the terms we want to express involve aspects of an object model. In particular, OCL provides a way to form expressions involving attributes and associations of objects. In addition, UML (and to some extent OCL) is becoming a standard.

The language describes object characteristics at the level of specifications. An implementation of this language requires a mapping between the classes at the specification and implementation layers. For a fuller discussion of implementation issues, see Section 3.17.

Note that not all features of OCL are currently implemented. In the following discussion we indicate both what has been implemented and some minor changes to the OCL language.

Notation
In describing grammar rules, we use the following notation:

- `*` means "zero or more"
- `+` means "one or more"
- `( )` means "a grouping"
- `?` means "optional"
- `a|b` means "choice of `a` or `b`"

3.1 OCL Language

This section describes the grammar and informal semantics for OCL expressions that will form the expression part of our policy language. Note that this usage of OCL is different from what is normally found in object modeling. A typical usage of OCL would be to form annotations on objects, such as invariants on object operations. We are using OCL as an expression language for expressing access rules – so OCL expressions are part of the data objects of the access control engine (see Section 3.18 for further discussion).

Note that we do not use the OCL syntax for "contexts". The context for all OCL expressions is the model corresponding to the access control evaluation engine, and not a particular class or function. (The elements associated with this "global" model are introduced in Section 6).

The descriptions presented in this section are based on the OCL specification report [OCL]. For a more complete description of the OCL language, see the OCL specification report. In Section 3.16, we describe some extensions we made to OCL.

3.2 OCL Comments

Comments start with two dashes and include everything up to the end of the line

```
-- sample comment
```

3.3 Types and Values

3.3.1 Basic values and types

The basic types are `Boolean, Integer, Real,` and `String`

Some examples of the possible values are:

- `Boolean: true, false`
- `Integer: 1,2, -38, 1243123, …`
- `Real: 1.3, 4.25, …` (integers are a subtype of reals)
- `String: 'Some text'`

Standard operations on these types

- `Integer: *, +, -, /, abs, div, mod, max, min, =`
- `Real: *, +, -, /, floor, abs, max, min, <, >, <=, >= ,=`
- `Boolean: and, or, xor ,not, implies, if-then-else, =`
- `String: toUpper, toLower, concat, size, substring(lower, upper), =`

(Strings operations are not currently implemented in our demonstration. Only arithmetic and comparison are implemented for Integer and Real.)

Mixed expression of integers and reals are allowed and integers are implicitly coerced into reals where appropriate (see [OCL] for more details).

Note that not all expressions in OCL are necessarily defined. An undefined subexpression in OCL does not necessarily mean that the expression is undefined. The boolean expression "false and anything" is false and the expression "true or anything" is true. However, for our purposes and unlike for standard OCL, "undefined and anything" and "undefined or anything" are always undefined.

In the slightly modified version of OCL that we define here, the handling for undefined terms with implication is similar to the above. The expression "false implies anything" is true. (We do not want the access rules to have side effects and a short-circuit method of evaluation can speed up the access control decision.)

3.3.2 Classes

The object introduction part of a model can introduce new classes (see Section 4.1). These class names can be used as types in the object expression language. Associated with a class is the structure consisting of its attributes (data members) and operations.

In OCL "=", "<>" are defined on user-defined classes to mean the objects are (or are not) the same. In our access control language this is the default interpretation. If needed, these operators can be defined differently in particular classes (see Section 3.16.4). For

example, it is possible to redefine "="to mean that the objects compared have the same attributes. However, to avoid potential user confusion it is recommended that if equality is to have a different interpretation that a different syntax be used. For example, one could introduce an operation `"byAttributeEqual"` to mean that attributes of two objects of some class have the same values. (Note that "=" for basic types, as opposed to classes, is defined to mean that the primitive values are the same.)

3.3.3 Enumerated types

The type of some variable may be a finite set of values. This is often called an enumeration type. The OCL syntax for an enumeration type is:

```
enum {value1,value2, ...}
```

In an OCL expression, a # is used before an enumerated value to avoid conflicts with attribute names. (See section 3.16.5 for a discussion of a potential alternative approach.)

3.3.4 The Let construct

OCL version 2.0 contains the "let" construct, which is useful for simplifying an expression. For example:

```
Let  user:actor = request.requestor in
   if  user.hasmasterauthorization() then
         true
   else if user.hasprojectauthorization() then
         user in {user1,user2}
   endif endif
```

is the same as

```
if  request.requestor.hasmasterauthorization() then
      true
else if request.requestor.hasprojectauthorization() then
      user in {user1,user2}
endif endif
```

This feature is currently not implemented.

3.4 Collections

3.4.1 Sets, bags, and sequences

Collections may be formed out of other types. Either the generic "`Collection`" keyword can be used or the more specific keywords: `Set`, `Sequence`, and `Bag`.

`Set{1,8,3}` is the set consisting of the elements `1`, `8`, and `3`.

Sequences have the elements ordered.

`Sequence{2,7,5}` is different from `Sequence{7,2,5}`

Bags are like sets, but allow duplicate elements.

`asSet`, `asSequence`, `asBag` can be used to convert between these collections.
Note that `asSequence` picks some unspecified ordering.

For integer expressions `a` and `b` with `a<b`, `Sequence{a..b}` is the sequence of elements starting at `a` and ending at `b`.

(Currently, only sets are implemented.)

In OCL, collections of collections are not the same as in ordinary mathematical usage. OCL collections are always flattened. `Set{Set{1,3}, Set{4,5}}` is the same as `Set{1,3,4,5}`. In the current version of the access expression language, collections are not automatically flattened. So `1` is not included in `Set{Set{1,3}, Set{4,5}}`.
To flatten a collection one could explicitly use a "`flatten`" operation. (No `flatten` operation is implemented in the current demonstration.)

3.4.2 Select, reject, and collect

`select`, `reject`, and `collect` can be used to form subsets.

These have not yet been implemented in the current demonstration.

There are three different ways to use these functions.
`collection->select(boolean-expression)`
forms the subset of "`collection`" whose elements satisfy the boolean expression.

`collection-> select(p | boolean-expression)`
also forms a subset. It is the set of all `p` in the `collection` that satisfy the boolean expression. This second form allows the iterator `p` to be evaluated in the expression

`collection->select(p:Type |boolean-expression)`
is like the second form but restricts the type.

A similar syntax holds for `reject`, where the expression describes the elements that are not in the set. (This is equivalent to selecting with the negation of the condition.)

`collect` is used to form subcollections of a different type from the base collection – for example, the set of ages in a collection of employees.

`a->collect(attr)` forms a bag of values of the `attr` attribute of each member of `a`. (Hence it is the same size as `a`.)

Other syntactic forms of collect are:

`a->collect(p | expression)`

`a->collect (p: Type | expression)` may also be used.

An additional syntax for collect is also available. When `b` is a set and `propertyname` is a property on items of that set, then

`b.propertyname` is a shorthand for `b->collect(propertyname)`

3.4.3 Additional collection operations

Additional operations can be applied to collections.

For any collection:

- `includes` (i.e., membership, `a->includes(b)` means that `a` "=" one of the elements of `b`)
- `size`
- `count`
- `includesAll` (i.e., superset, `a->includesAll(b)` is true when `b` is a subset of `a`)
- `isEmpty`
- `notEmpty`
- `sum` (i.e., add all of the elements of the collection)
- `exists, forall, iterate` (see Section 3.10)

(Only the includes and includesAll operations have been implemented.)

In addition to these OCL operations, we introduce a `flatten` operation to explicitly flatten a collection. (Recall that we are not automatically flattening collections, in contrast to OCL; see Section 3.4.1.)

For sets and bags:

- `union`
- `intersection`
- `=` (for sets they have the same values. For bags, the values must occur the same number of times.)
- `-` (set difference)
- `including` (adds an element to the set)
- `excluding`
- `symmetricDifference` (for sets only)
- `select`
- `reject`
- `collect`
- `count` (number of occurrences of an object)
- `asSequence` (conversion to a sequence picks some order for the elements)
- `asSet, asBag` (conversions for bags and sets, respectively)

For sequences:

- `union` (sequence concatenation)
- `=` (the sequences have the same elements and in the same order.)
- `append` (add one new element)
- `prepend`
- `subSequence`
- `at` (value at ith position)
- `first`
- `last`
- `including` (same as `append`)
- `excluding`
- `select`
- `reject`
- `collect`
- `iterate`
- `asBag`
- `asSet`

None of these operations has been implemented in the current demonstration.

3.5 Object Properties

Object attributes, object operations, and object association ends are collectively referred to as object properties.

3.5.1 Object attributes

We refer to an attribute of an object with the period notation. For example, attribute `b` of object `o` can be referred to as `o.b`.

The OCL expressions that are used here are in the context of the access control engine. The term "`self`" in a rule refers to that object. Typically, this term is dropped because it is clear from the usage. (The "`self`" keyword has not been implemented in the current demonstration.)

Properties of a set of elements are indicated by using an arrow. For instance, the expression "`a.start->size`" specifies the size of the set "`a.start`".

When an object is not a set, the arrow operation treats the object as if it were a singleton set.

3.5.2 Object operations

A class may also introduce operations. The period notation is also used to indicate invocation of the operation.

For example, `a.f(c)` is the operation `f` applied to `a`, with additional parameter `c`.

3.5.3 Object association ends

Classes can be related in a number of ways in UML. An "association" between two classes represents the particular relationships between the instances of those classes. (In UML, there are other kinds of relationships between classes that do not correspond to relating the instances of the classes, for example, generalization.)

We use associations as part of access control expressions. Not all of the UML relationships between objects of a specification will correspond with relationships in the expression language (relationships that are intended to be evaluated). Instead, the relationships of the expression language correspond to those relationships that will be asserted with credentials.

In addition to having a name, an association may have "source and destination" names, to navigate from one end of the association to the other.
These are called the "association ends" of the association.

Suppose `R` is an association starting in class `C` and ending in class `D`. Suppose further that the association end names are "`start`" and "`final`" for class `C` and class `D`, respectively. Say `c` is an object of class `C`.

Then "`c.final`" refers to the elements `d` of class `D` for which `R(c,d)`. In OCL, "`c.final`" is a set of values in `D`, unless the size of "`c.final`" is constrained to be at most one element (see next paragraph). In this case, "`c.final`" is just an element in `D` and not the singleton set of that element.

Sometimes association end names are also called "roles". We avoid the use of the word "role", since in the context of access control, this may be confused with user roles. Also, some authors refer to the destination of an association as the "target". We avoid this convention, because we use the word "target" to refer to a target object.

A multiplicity constraint in UML describes size limitations on an association. The syntax "`a..b`" indicates the range of allowable sizes for an association end. Probably the most useful forms of multiplicity allow a user to indicate whether a relation is a function, partial function, and whether it is 1 to 1. In the present version of our system, we only support two cases. Either the multiplicity is not specified or the destination multiplicity is indirectly specified as 0..1 from a dynamic attribute declaration (see Section 4.1.4).

When an association end name is not specified in the UML description of an association, the end name defaults to the association name, with the first letter in lower case (provided there is no ambiguity).

3.5.4 Navigation to and from association classes

Navigation to an association class (see Section 4.1.3) also uses the dot notation. The expression `o.a`, where `o` is some object and `a` is an association class name with the first

letter lower case, means the set of all objects of the association class that are involved in the relationship from object `o` by the association `a`.

If `o` is an association object (i.e., link object) with an association end named `r`, then `o.r` is the object at that end of the relationship indicated by `r`.

Our current implementation does not support navigation to or from association classes.

3.6 Casting and Conformance

Certain types are automatically changed to other types before an operation is applied. For example, an integer may be converted to a real in the evaluation of an arithmetic expression. In fact this is the only case that is currently supported.

Types can be explicitly recast, when appropriate, by using:

```
someobject.oclclasstype(SomeType)
```

Currently type recasting is not supported.

For details about type casting and conformance, see the OCL reference manual [OCL].

3.7 Operator Precedence

The precedence of the OCL operators is

- '.' and '->'
- unary `'not'` and unary '-'
- '*' and '/'
- '+' and binary '-'
- `'and'`, `'or'`, and `'xor'`
- `'implies'`
- `'if-then-else-endif'`
- `'<'`, `'>'`, `'<='`, `'>='`, and `'='`

Note that arithmetic operators follow the normal mathematical convention, however, `and`, `or`, and `xor` have the same precedence. This operator precedence is built into the OCL grammar (see Section 3.14).

Also note that the precedence for `implies` is ambiguously specified in the current OCL documentation. The textual specification in [OCL] is as above, but the OCL grammar [OCL] differs. (In the current implementation, the operator precedence for implies is as above.)

The current implementation also splits the precedence level for `and`, `or`, and `xor`. The order from lowest to highest is `or`, `xor`, and `and`.

3.8 Packages

In UML, types may be organized into packages.

`Packagename::typename` can be used to specify the type in a package

The current implementation does not support this feature.

3.9 Type Features

There are additional OCL features pertaining to using OclType.
These include `oclType`, `oclIsTypeOf`, `oclIsKindOf`, `oclAsType`, and `allInstances`. They are currently not supported.

- `oclType` returns the type of an object.
- `oclIsTypeOf` returns true if the type of the object is the specified type
- `oclIsKindOf` returns true if the type of the object is the specified type or a supertype of that type
- `oclAsType` allows an object to be treated as if it was in a supertype to get at overridden attributes.
- `allInstances` returns the set of all instances of a given type. The OCL reference manual recommends that this feature not be used.

`OCLExpression`, `OCLType`, and `OCLAny` are additional basic "types" in OCL. They are currently not supported.

- `oclExpression` is the "type" of an OCL expression
- `oclType` is the "type" of types
- `OCLAny` is a supertype of every type

oclIsNew and oclIsInState are not applicable to the way OCL expressions are being used and are not supported.

3.10 Quantifiers

Quantifiers, such as forall and exists, can be used to form complex expressions. This feature is best avoided when writing access rules. First, the computations involved may be time consuming. Secondly, the information available at the information server may only partially capture the real-world situation, and hence a computed interpretation of the expression may not produce the expected value (see Section 3.16.3). When possible, the application developer should provide methods that reduce the need for quantifiers. (Quantifiers have not been implemented in the current demonstration.)

The expression `c->forall(p1,p2 | p1 <>p2 implies p1.name <> p2.name)` results in `true` if, for every `p1` and `p2` in the collection `c`, if `p1` is not equal to `p2` then `p1.name` is not equal to `p2.name`. That is, the expression is true if different elements of `c` have different names.

In OCL, there are also two additional syntactic forms for the forall quantifier.

Similar constructs are available for the "`exists`" quantifier.

3.11 Iterate

The "`iterate`" construct provides another way of making quantified assertions.

```
a->iterate(elem:Type; acc:Type = expression | expression-with-elem-and-
acc)
```

`acc` is the accumulator of the construct and is originally set to `expression`.
The expression `expression-with-elem-and-acc` is evaluated for each `elem`, as `elem` iterates over the collection `a`. Each time through the iteration, the accumulator is set to the value of `expression-with-elem-and-acc`. The result is the final value of `acc`.

The following Java-like pseudocode describes the semantics.

```
iterate(elem:T; acc:T2=value)
{
   acc = value;
   for (Enumeration e= collection.elements(); e.hasMoreElements;)
   {
      elem=e.nextElement();
      acc=expression-with-elem-and-acc
   }
}
```

Note that the value of the result may be implementation dependent for sets and bags if the order of evaluation makes a difference. As with quantifiers this feature has not been implemented in the current demonstration

3.12 The @ Operator

Time expressions in OCL are indicated by the `@`. This can be used for describing the value of an expression before or after a method is invoked (in a precondition or postcondition). However `@pre` in OCL is not particularly useful for access rule expressions and is not supported in our language.

3.13 Context

In OCL, the context syntax is used to designate the class or operation to which a particular invariant or condition applies. For this project, OCL expressions are used for access rule evaluation and not as constraints on classes or operations.

A "context" for the ACL expressions could be considered to be the access control engine.

3.14 Grammar as in OCL manual

```
expression := letexpression? logicalExpression

logicalExpression :=
   relationalExpression ( logicalOperator relationalExpression )*

relationalExpression :=
   additiveExpression ( relationalOperator additiveExpression )?

additiveExpression :=
   multiplicativeExpression ( addOperator multiplicativeExpression )*

multiplicativeExpression :=
   unaryExpression ( multiplyOperator unaryExpression )*

unaryExpression := ( unaryOperator postfixExpression )
             | postfixExpression

postfixExpression := primaryExpression ( ("." | "->") featureCall )*

primaryExpression := literalCollection
            | literal
            | pathName timeExpression? qualifier?
                       featureCallParameters?
            | "(" expression ")"
            | ifExpression

letExpression := "let" <name> ( ":" pathTypeName)? "=" expression "in"

literal := <STRING> | <number> | "#" <name>
literalCollection := collectionKind "{" expressionListOrRange? "}"

expressionListOrRange :=
   expression ( ( "," expression )+ | ( ".." expression ) )?

featureCallParameters := "(" (declarator)? ( actualParameterList )? ")"

featureCall :=
   pathName timeExpression? qualifiers? featureCallParameters?

ifExpression :=
   "if" expression "then" expression "else" expression "endif"

enumerationType := "enum" "{" "#" <name> ( "," "#" <name> )* "}"

simpleTypeSpecifier := pathTypeName
            | enumerationType

qualifiers := "[" actualParameterList "]"
declarator := <name> ( "," <name> )* ( ":" simpleTypeSpecifier )? "|"

pathTypeName := <typeName> ( "::" <typeName> )*
pathName := ( <typeName> | <name> ) ( "::" ( <typeName> | <name> ) )*
```

```
timeExpression := "@" <name>
actualParameterList := expression ( "," expression )*
logicalOperator := "and" | "or" | "xor" | "implies"
collectionKind := "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator := "+" | "-"
multiplyOperator := "*" | "/"
unaryOperator := "-" | "not"
typeName := "A"-"Z" ( "a"-"z" | "0"-"9" | "A"-"Z" | "_")*
name := "a"-"z" ( "a"-"z" | "0"-"9" | "A"-"Z" | "_")*
number := "0"-"9" ("0"-"9")*
string :=
      "'" ( (~["'","\\","\n","\r"])
            | ("\\"
                  (     ["n","t","b","r","f","\\","'","\""]
                        | ["0"-"7"] ( ["0"-"7"] )?
                        | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                  )
             )
          )*
"'"
```

3.15 Constants

Associated with OCL constants are lexical rules for how they are recognized.

Integer constants are the standard decimal representations of numbers (The lexical rule is actually part of the OCL grammar as the definition of a `number`.)

Boolean constants are the names "`true`" and "`false`".

Real constants are decimal numbers. (We also support real constants that have an exponent. We use the lexical rule of Java.)

Enumerated type constants are names.

The constant "`NULL`" is a possible value for any object and indicates the absence of an object reference. (See section 4.1.4 for a case in which it is useful.)

Collection constants, such as Set{3, 4, 5}, are parsed as in OCL.

3.16 Language Extensions

3.16.1 Set operations

We augment the language with the convenient syntax "`x in a`" for the expression "`a->includes(x)`".

We also introduce the syntax "`a contains x`" for the expression "`a->includes(x)`".

The precedence of these options is in the following place in the hierarchy. (See Section 3.7 for the full hierarchy.)

- '.' and '->'
- 'contains' and 'in'
- unary 'not' and unary '-'

Note that in the current implementation, the "contains" and "in" operations have been placed at the precedence of relational operators, rather than at the level of '.' and '->'.

3.16.2 Names of users and files

We would like a convenient syntax to refer to individual users in expressions. Since users in different organizations may have the same internal "name", we need to use "namespaces" to distinguish them. We could introduce a type of `Sequence(NameSpace,String)`, but the syntax for designating a name would be inconvenient, especially as it may appear frequently. Hence, we intend to use a syntax such as:

```
username := Namespace

Namespace := name ( ("from"  NameSpace) | ("/" Namespace) )*
```

It is possible that a future version of the system might also support `name :: NameSpace` as this is how package names are built in OCL.

We avoid the standard syntax of periods (such as in email addresses), because the period is already used in obtaining the attribute of an object and invoking an operation. Also, the forward slash is sometimes used for directory structures, and it is also used in Adage.

Note that a user GUI could use shorter names without a fully qualified namespace, if the prefix of the name was the same as the users.

Namespaces are not currently implemented. Note that it may also be useful to use namespaces for files. This is also not implemented.

In the current design, not all names used in an expression must resolve to references to objects. This accommodates the fact that both users and files are not necessarily persistent for the lifetime of an information sharing project. At the time an OCL expression is evaluated, if a name does not refer to some object then the result is the null object. Thus, a name is essentially an abbreviation for a function of the form `getobject("somename")`, i.e., a function that returns an object based on some string. This has the advantage of making the syntax simpler to read, but the disadvantage of

having weak typing. An alternative syntax could explicitly require something like Getuser("somestring"), whenever the type was not inferable from the context. Users can avoid this use of names for objects by referring to a string attribute of an object in a rule rather than to the object itself. E.g., one could use

```
if (request.requestor.name= "Fred")
```
rather then

```
if (request.requestor=Fred).
```

Since a name can refer to a null object, expressions should be built that check for null before trying to reference an attribute of an object referred to by a name.

In the current design, we do not enforce the OCL restriction that names of objects should start with a lower case letter and names of types should start with a capital letter. (For example, the name `Fred`, can refer to a user.)

3.16.3 Interpretation of expressions involving relationships

We may need to refer to the manner in which a particular relationship between objects should be determined.

Consider the expression

```
not (a.employedby contains b)
```

What should the expression mean?

- the requestor does not provide "`employedby`" evidence
- the information server does not have the evidence
- some attribute authority does not provide such evidence
- there is no evidence
- the real-world `employedby` relationship does not hold between `a` and `b`

In practice, there is no way to evaluate the last item, because the computation must be based on some data, but the other alternatives might be useful.

We could introduce a new keyword, such as `eval`, that specifies how a particular expression should be evaluated.

```
eval(how_to_evaluate_id, expression)
```

However, no such extension is currently implemented, and we currently interpret relationships by using the evidence at the information server.

3.16.4 Comparison operations

It would be convenient to use some of the standard operation symbols for classes other than the basic types. The most useful, for our purposes, are the comparison operations. (These can be introduced in new classes, see Section 4.1.1.1.)

If operators are defined for either "<" or "<=", and operators are not defined for the other inequalities, then they will be implicitly defined using comparison methods in the standard way. (Note that we would not normally expect someone to define both ">" and "<".) If the operator "=" is defined and "<>" is not defined, then "<>" is implicitly defined as not "=".

As Java does not support overloading these operator symbols, the implementation convention is that the names of these symbols are called: `equals, notEquals, lessThan, greaterThan, lessThanOrEqual, greaterThanOrEqual.`

(In the current implementation, only `greaterThan` and `Equals` should be defined. The others are implicitly defined by these.)

3.16.5 Enumerated values

It may be useful to weaken the rule that requires the use of # before an enumerated symbol in an expression. The "context" for access rules is the access control engine, and it is possible that an enumerated value might conflict with a named object (e.g., the attribute of the access rule evaluation class need not be prefixed with `self` in an access control rule). An alternative to the use of the # is to simply prohibit the use of enumerated values which will conflict with those names.

Since enumerated values have not been implemented, this recommended change has no impact on the current demonstration.

3.16.6 Extension for collection type formation

We introduce a generalization of simpleTypeSpecifier to handle collections.

```
TypeSpecifier := pathTypeName
               | enumerationType
               | collectionType
               | "AnyType"    -- note the standard OCL keyword is OCLAny

collectionType := collectionKind "(" TypeSpecifier ")"
```

The grammar for introducing classes uses `TypeSpecifier` when introducing attributes.

It is possibly useful to introduce new type names for types that are built from collections and enumerated types. (See Section 4.1.5).

3.17 Implementation Issues

3.17.1 Class definitions

The policy language allows new classes to be added, so the access control engine can be designed to evaluate a wide range of expressions. For example, an integrity level class could be introduced and an integrity level attribute could be part of a target class.

Note, however, that for the interpreter to evaluate expressions that involve new classes, additional executable classes are required that implement the desired functionality of those classes. If a new concept of integrity level is desired, and comparisons of these levels are used in an expression, then a definition of the comparison functions must be provided in a Java class.

Java implementation classes contain sufficient information so that a separate specification of these classes is not needed. Information about the members and methods of a "new" class can be obtained at run-time. However, some kind of specification is needed in order for users to understand the terms. We provide a syntax for textually specifying new classes in Section 4.1.

3.17.2 Language constructs

For uniformity in handling the objects in the Java implementation, we have implemented real, integer and boolean as the classes Double, Integer and Boolean respectively (as opposed to the primitive types). Similarly, we use Java Strings for Strings.

For an OCL class, we use a Java implementation class with fields corresponding to the attributes of the OCL class and methods corresponding to the class operations. This use of Java classes has the side effect of mingling Java functionality with OCL functionality. For example, there will be methods on the Java class intended only for internal system use, and which are not to be used in OCL expressions; the same holds true for attributes. There is no mechanism presently in place to prevent a rule from invoking a method on the Java object that is not intended to be part of the OCL notion of the rule.

3.18 Language and Meta-Language

This section provides some technical points for UML modelers on the difference between using OCL in the access control system presented here and the more standard usage. (Other readers can skip this section.)

OCL is typically used to express properties on functions or classes, such as invariants, preconditions, and post-conditions. Thus, OCL is used as a "meta-language" to make assertions about the functions of an object. An OCL expression as used here, however, is data that is evaluated by (Java) code. Thus, certain constructs that are reasonable to use as invariants are less useful for specifying a term to be evaluated. Generally, the OCL used here is for the purpose of "navigating" to the attributes of some class and then applying Java methods to those values.

Readers experienced in UML modeling should note that because of the difference in how OCL is being used, what constitutes specification and what constitutes implementation with our access control mechanism are not standard. The operations that will be evaluated, from the point of view of access control, are those used in determining object characteristics. The operations that are only specifications (and do not get evaluated during access control) are the real operations to be applied to the target objects.

3.19 What follows

So far, we have just introduced how to form expressions in the language. Some of the identifiers refer to classes, relationships, and attributes. In the next section, Section 4, we describe a syntax for introducing these classes, relationships, and attributes. Then, in Section 5 we will show how these expressions are used to construct security policies.

4 Classes, Relationships, and Attributes

In this section, we describe a syntax for describing the classes, objects, and relationships that are used in the expression language. This part of the specification is needed so that users can understand the vocabulary used in the access control expressions. It need not directly correspond to the class definitions used in the access control engine implementation.

Note that a graphical, UML style of specification could be used instead of the textual specification presented here.

4.1 Object Model Declarations

As in UML, there are classes (with attributes and operations) and relationships. These are declared by `classDecl` and `relationDecl` constructs, respectively.

There are also declarations for specific objects and values, `objectDecl` and `valueDecl`. We also use `attrDecl` statements for specifying attributes that are defined at a later stage than object creation (similar to relations, see section 4.1.4). Finally, we permit the introduction of aliases using `AliasDecl`.

```
objectmodelDecl := classDecl |objectDecl | valueDecl | relationDecl
                           |attrDecl | AliasDecl
```

4.1.1 Class statements

Class statements specify classes, their attributes, and their operations.

In our approach, there are two different kinds of classes. One kind describes data types used in evaluating terms in the access rules. The other kind characterizes the methods and attributes of the target objects. Determining access to a target object should not involve applying one of the controlled operations on the target, since an access decision should be made prior to any action taken on the target. However, in the access rules, we may want to refer to some property of an operation on a target class (e.g., performs READs but not WRITEs). To clarify this distinction, we will introduce a syntactic difference to distinguish between these kinds of classes. They are "`Class`" and "`TargetSpecClass`", respectively. Classes designated by "`Class`" are referred to as standard classes.

```
classDecl := standardClassDecl | targetSpecClassDecl
```

4.1.1.1 Standard classes

Aspects of a class that may be defined are described below.

- *Attributes.* Attributes can be shared among all objects of a class, or they can be object members. We use the word "`shared`" to indicate class attributes that are shared among all the objects of a class. (In programming languages, this is sometimes called "static". In UML, it would be called "class-scope".)

- *Operations.* Operations may be invoked on the objects of the class. These are used both in the specification of access rules (part of the OCL expressions), and they are realized in an implementation to do part of the access evaluation. (In UML, methods are the implementation of an operation. We currently use the keyword "`Operation`" to introduce the signature of an operation/method. We currently use the keywords "`Operation Symbol`" to introduce symbols involved in operator overloading.) Class operations, i.e., those that do not need to refer to a specific object of a class are indicated by the keyword "`ClassOperation`". (In Java, such constructs are called static methods.)

- *Inheritance.* A class may inherit from another class. (Only single inheritance is permitted.)

Named objects of particular classes can be introduced as part of the application class (see below 4.1.2).

```
standardClassDecl := "Class" classname
            (
                   ("Inherits" pathName )
                  |  ("shared")? attrname ":" attrtype
                  |  ("Operation" | "ClassOperation") opnamedef
                        "("  operation_parameters ")"
                        (":" returntype)?

            )*
            "End"

classname := name
```

Operations can have an alphabetic name or can refer to a comparison operator.

```
opnamedef := name |
            "Symbol"   ("="|"<>"|"<"|">"|"<="|">=")

operation_parameters :=  operation_parameter*
operation_parameter := paramname (":" typeexpression )?

returntype := <name>

paramname := <name>

attrname := <name>
```

```
attrtype := TypeSpecifier
```

It would be desirable to use packages to aid in grouping related classes. This is used in OCL but our current demonstration implementation does not support it.

4.1.1.2 Target specification classes

A target specification class is used to model characteristics of target classes needed for access evaluation. Its declaration syntax resembles the standard class declaration, but its use is different.

The key differences between a target specification class and a standard class are:

1) a target specification class corresponds with a class of the object manager's target objects, and
2) a target specification class defines "actions", and properties of those actions, which correspond with the methods of the object manager's target objects. (Note that this notion of action is not the same as the UML notion of action -- the actions of a target specification class are a kind of attribute used to characterize a method of another class.)

Note that every target specification class has the same name as some target class of the object manager.

Item (2) permits access rules to be expressed in terms of properties of the methods that are to be invoked on an object. A method of a target class is represented as an "action" in the target specification class. An action can have its own attributes (called "properties"), and access rules can involve these properties. An access rule may also involve the arguments to be supplied to a method of the actual target object. These are treated as attributes of the action. Return types of actions are not used, since the values returned by the actual operations will not be known at the time of the access decision.

The collection of all actions constitutes the objects of type `Actiontype`. This type is not part of a particular target specification class. This permits expressions such as `request.operation = read`. However, in the current version, because actions are associated with the methods of some target object, they are incrementally specified in their appropriate target specification classes. If an action is introduced in more than one target specification class, it refers to a common action and must have an identical specification. (The current implementation does not permit the same action in more than one target specification class; each action of a given target specification class must have a globally unique name.)

Suppose `newfile(init:String):boolean` is a method of a target class, `directory`, where `newfile` has a property called "`access`" with possible values of `READ`, `WRITE`, or

BOTH. The newfile action would be specified in the target specification class, directory, as

```
Type accesstype = enum{READ,WRITE,BOTH}

Action newfile(init:String) property access:accesstype is WRITE
```

This introduces an action object for the class that can be referenced by directory.newfile. The object has an attribute, called opname, that is the string that names the operation, in this case "newfile". It has an attribute, access, with its value set to WRITE. It also has an attribute for the parameter init of type String, which is defined at the time a request is made. (A default value for init could be used for requests with unspecified parameters. This feature is currently not part of the specification language.)

Note that in the current design, all parameters of a request are contained in a parameter list. There is no explicit specification that associates the name of the parameter for the action with a parameter in the parameter request list. In a future version of the system, we may switch the parameter list to a list of parameter name/ parameter value pairs. In this case, the name of the parameter in the action operation should correspond to one of the parameter names in the list of pairs. In the current version, there should be an informal description that describes the purpose of each parameter in the parameter list.

When a request is made, it will include the operation name and possibly parameter/value pairs. This introduces an action object of the type described above, with the parameters of the action object set to the values of the request (possibly using the defaults of the action object for the class). "Property" values of the action object associated with the request are set to the property values of the action object of the target class.

There is one built-in property for all actions, which indicates whether a particular action is a kind of object "creation". This is called "isCreate" and it is either true or false. If the property is not specified, it is assumed to be false. For files and directories, these are the operations "CREATE" and "CREATEDIR" respectively.

One can form expressions involving the action, such as:

```
(request.operation.access = READ)
(size(request.operation.init)< 1000)
(request.operation.opname=directory.newfile.opname)
```

A target object will also have trust polices associated with it. These are described in Section 5.

```
targetSpecClassDecl := "TargetSpecClass" classname
          (
              ("Inherits" pathName )
            | ("shared")? attrname ":" attrtype
            | "Method" opnamedef  "(" operation_parameters ")"
```

```
            (":" returntype)?
        |   "Action" opnamedef
            ( "("  operation_parameters ")")*  (op_property)*
        |   "Actions" (opnamedef)* "End"  -- an alternate notation
                                          -- for listing many actions
    )*
    "End"


op_property := "Property"  name ":" attrtype is expression
```

In the current version, if no inheritance is specified, it is assumed that a particular `TargetSpecClass` inherits the class "`TargetSpec`". See Section 6.2 for a description of the `TargetSpec` class.

Note that actions are really not part of the target specification class. They incrementally define the set of actions in an enumerated type called `Actiontype`. When inheritance is used, the set of actions "for a target specification class" is the union of the actions from the inherited class with the ones introduced in the target spec class. The way actions are incrementally defined has no impact on access rule evaluation, but, for clarity, the actions for a target specification class should correspond to the operations that can be applied to the corresponding target class.

Note that the specification of a target class does not provide details such as how the attributes are set or retrieved. For example, the creation date may be associated with a file, so that retrieval of such an attribute may involve a file system call.

In addition to the attributes associated with a target specification object, there are access policies that govern the access control to that object. See Section 5.

Actions are currently only implemented in a simple form. The present demonstration implementation does not provide support for handling action attributes.

4.1.2 Object declarations and default objects

Sometimes access rules must refer to particular values—that is, to objects of the access control engine class. Some of these are constant for the application, and some are set in the course of the system being used.

For example, one of the predefined objects described in Section 6.4 is the "`request`".

These named objects are essentially attributes of the access control engine class. These attributes can be formally introduced using the class syntax described above.

If all or part of an object is really constant for the application, it might be useful to specify the values of the constant parts. We introduce a syntax to support this.

```
objectDecl:= "Object" name ":" name
```

```
        (attrname "is" OCLexpression)*
    "End"
```

The type of the attribute referred to by `attrname` is defined by the class of the object. The OCLexpression on the right should indeed have that type.

Non-object values can also be introduced.

```
valueDecl := "Value" name ":" typename is OCLexpression
```

where the type name is either a basic type or a collection operator.

4.1.3 Relation statements

We introduce a syntax to declare a "relation" (called an "association" class in UML [Rat97c]) that encapsulates a relationship between two classes, and also may carry attributes. For instance, one may have a relation "ReportsTo" between the classes "Actor" and "Actor", with a boolean attribute "supervisory". An instance of a relation (called a "link object" in UML) is therefore a pair of objects of the respective classes, along with attribute values.

The UML/OCL association syntax provides a convenient way to build expressions that can be used to go from source to destination and from destination to source. Note that in UML, an association class cannot possess two different link objects with the same source and destination and different attributes.

We may want to use certificates signed by some authority to indicate whether a given pair of objects is in fact related. We can use a relation instance to store the trust information that the certificates convey.

In addition to supporting certificate based relationships, we may need to refer to relationships that are built into the structure of the target of the objects. For example, the fact that a file is "contained in" a directory will be a relationship that is implicitly defined by the file system, and will not be certificate based. We use the keyword, `builtin`, to indicate those relationships that are not certificate based.

The syntax for relations is:

```
relationDecl :=  "Relation" name ("builtin")?
                 "Source" classname (endname)? (multiplicity)?
                 "Destination" classname (endname)? (multiplicity)?
                 (relationattr)*
                 "End"
```

The "Source" and "Destination" keywords are used to introduce the "association ends" (see Section 3.5.3). (In order to avoid ambiguity between navigating to relation instances and source (or destination) objects, association endnames should be specified.)

```
endname := name

multiplicity := "Multiplicity" multexpression ".." multexpression
multexpression := "*" | expression

relationattr :=  "Attribute"
                (   attrname ":" attrtype)*
```

(An alternative syntax would be to replace the collection of relation attributes with a class that contains those attributes. The disadvantage of this approach is that it makes the expressions that refer to the attributes more complex.)

In the current version of the project, the multiplicity constraint is not used. If a destination constraint of 0..1 is desired, then the association should be specified as a dynamic attribute (see Section 4.1.4).

In UML, it is possible to indicate navigability constraints as to how a relation (link object) can be obtained. In particular, it may only be possible to obtain the destination object from the source, but not vice versa. (In the UML graphical language, this is represented using arrows on the association line.) This specification is not supported in our current language. However, an application implementer may choose to implement only one-way lookup for some associations. In this case, an informal comment should be added to the specification.

It is possible that there are link objects with particular attributes that a user should be aware of. One could use the following syntax:

```
"Relation Instance" name name
                                -- relationname instancename
         (attr "is" value)*
"End"
```

Relationships can be useful in building groups. For example, a relationship could be established between an employee, A, and his supervisors, say, "ReportsTo". Then the expression manager `A.reportsto` would specify the group of users to whom A reports. As another example, there could be a class of objects called projects and a relationship, `members`, that specifies the members of the project. The expression `collaboration_proj.members` would indicate the members on that project. The authorization authority for establishing the project members need not be a system administrator. The appropriate signatures on the certificates to establish membership can be defined in the trust rules for that relationship.

The grammar supports relationship attributes, and they are needed if we want to refer to the trust of some particular association instance. However, using OCL to refer to the attribute is awkward. Suppose `object1` is connected to `object2` by `samplerelation` and that there is a link instance associated with this relation. In OCL, `object1.samplerelation` refers to the set of all link objects that `object1` is connected to. One could take the intersection of `object1.samplerelation` and `object2.samplerelation`, to obtain the set of link objects (there is at most one). `object1.samplerelation ->intersect(object2.samplerelaton).`

We introduce an extension to the OCL syntax to handle this.

```
"therelation" "(" relationname "," objectname  "," objectname  ")"
```

This specifies the partial function that returns the link object between the objects. For example, an attribute "temperature" of a link object might be obtained as:

```
therelation(samplerelation,object1,object2).temperature
```

This is not implemented in the current demonstration.

As an alternative, we might have the return value be a set that is either empty or a singleton set, in order to make the function total.

4.1.4 Dynamic attribute declarations

Some attributes associated with an object might be provided as independent evidence from the object itself, such as by a certificate signed by some authority. We refer to attributes based on certificates as dynamic attributes.

The general notion of a dynamic attribute is that it is a value attested to by some certificate. From an implementation point of view, such an attribute is like the destination of a (functional) relation. From a conceptual point of a view, it is just an attribute of an object. However, if we want to provide trust information about associated attributes, then a pure attribute approach becomes messy. We would need the attribute to contain both a value and certificate information, and then we would have to use a (secondary) attribute name to reference the value part of the attribute. For this reason, we will introduce a new syntax.

A simpler notion of dynamic attribute that might also be useful is one that indicates the existence of a certain certificate (where the value is not important). For example, a certificate could be used as a kind of token to grant a capability to a user. Note that this is different from an ordinary attribute whose value is a certificate, in that the dynamic attribute is assigned based on the receipt of some certificate, and therefore is not determined at the time the object is created. This distinction is a reason to introduce a different syntactic construct for this kind of dynamic attribute.

The current specification language supports both of these notions of dynamic attribute. It allows dynamic attributes with or without "values". The information from an external

certificate is divided into two pieces, a piece that is used for all dynamic attributes (the common certificate type) and a value piece (when appropriate).

We use a modification of the relation syntax for dynamic attributes.

```
attrDecl :=  "Attribute" name
                  "Source" classname
                  ("Destination" classname (endname)?
                        ("default" OCLexpression)?)?
                  "End"
```

The expression o.a, where a is the name of the attribute, is the representation of the type of dynamic certificate if one exists, otherwise it is NULL.

The destination field is optional (in case the value is not needed). Unlike relations, the expression `o.d`, where `d` is a destination of a dynamic attribute of `o`, returns an element of the type of the destination class (the "value"), just like a normal attribute of a class (relations, in contrast, return sets of values). Note that this formulation of dynamic attributes is equivalent to UML associations with multiplicity 0..1.

If an attribute value has not been specified for an attribute with a destination field, then the attribute will have a default value. If a default value field is present in the specification then that field defines the attribute's default value. If the field is not present and the attribute type is a class then the default value will be NULL, otherwise, the attribute type is a basic type and the default values are as follows:

- Integer is 0,
- Real is 0.0,
- Boolean is False, and
- String is "".

Value-based attributes are not implemented in the current demonstration. That is, destinations are not used. Also, there is currently no user interface for adding, viewing or deleting dynamic attributes.

4.1.5 Aliases

It may be useful to have aliases for objects or values that are defined by an expression. For example, we might want to refer to `a.b.c.d` as just `e`. This is particularly useful when unwrapping the contents of a request. Aliases can be introduced by:

```
"Alias" name expression
```

New abbreviations could also be introduced for collection and enumerated types

```
TypeDecl "Type" name (<collection-type> | <enum-type> )
```

These abbreviations are currently defined to be global. An alternative would be to also allow them to be used locally in the context of some class.

These two features are not implemented in the current demonstration.

4.1.6 Simple examples of classes and relations

```
Class Actor  -- representing users of the system
   inherits base_actor
                  -- we describe base actor class in Section 6.1
   jobtitle : text
End

TargetSpecClass File  --Targets are described in Section 6.2
   action update() property access:accesstype is WRITE
End

Relation ReportsTo
   Source Actor manages
   Destination Actor reportsto
   Attribute signedby employer
End

Relation Owns
   Source Actor ownedby
   Destination Target owns
End
```

4.2 Context

The syntax described above can be used to describe the UML model context. Although such a specification is not required in order to build an access control engine it is important that such a specification exist because users of the system need to be able to understand the vocabulary that is used in the formation of the access rule expressions.

In Section 6, we will provide informal descriptions of the classes and objects that are expected to be part of the context. These will be made more precise with a specific example application, see Section 7.

5 Policies

In this section, we describe how trust and access control policies are formed.

5.1 Combining Policies

Because, in general, there will be multiple parties with a stake in protecting a given set of resources and information, there is a need for combining multiple policies on a given target.

These might include:

- Administrative policy – administrators of the information (who may not necessarily even be users of the information sharing application) may impose rules
- Rules-of-the-game policy – what all participants of the information sharing activity agree to, or sign up for.
- Creator/owner policy – how the creator of a particular object wants it to be controlled.
- Intermediate controller policies – There may be intermediate policies inherited from resources in which an object is embedded. For example, the "owner" of a directory may impose certain constraints on all subdirectories (even when they are "owned" by a different user.)

To gain access to a target object, a request must satisfy all of these policies. Access will be made sufficiently restrictive so that all parties' restrictions hold. (It is possible that this could limit certain functional objectives of some users. Either they will have to get other parties to agree to some change, or certain kinds of information sharing will have to be handled under a different arrangement.)

Each of these policies is currently constructed as conjunctions of disjunctions (a set of rules containing subrules). Note, however, that rules can be arbitrary boolean combinations of conditions, so there is no inherent limitation on the expressiveness due to the current mechanism.

Eventually, we would like to support the selection of policies under special circumstances. For example, an owner of a directory may invoke a privilege to remove a subdirectory controlled by a different owner. (See Section 5.6.4.) This is currently not supported.

5.2 Control and Modification of Policies

Parties may want to change their policies over time. There are a number of issues associated with this.

The first issue is who gets to modify the access rules and access attributes. This could be handled by using modifiable OCL rules to define who is allowed to view or modify a policy. However, if the view/modify rights are modifiable, then an "owner" could set the modify rules so that the owner is not allowed to make changes. In this case, special circumstances might have to be invoked for the owner to update the policy. Another example of a problem with this approach is that not all access attributes should be modifiable by the creator of an object, e.g., the date of creation. In the current design, we partially get around some of these difficulties by allowing application level view/modify rules to take precedence over view/modify rules of a policy. (In fact, in the current

implementation of the demonstration, policy view/modify rules are not used.) A different alternative is to use a more expressive meta-policy mechanism. However, we are concerned with controlling the complexity of the system and so have not followed this path.

The second issue deals with propagation of changes. If a policy on a target is imposed on subtargets, then is a change on the target's policy also propagated to subtargets? If changes automatically propagate, then the controllers of subtargets may find that the restrictions they thought they were imposing are no longer adequate, or certain types of sharing may no longer work properly. Alternatively, if there were no propagation, then the controller of a higher-level resource would have less discretion on how that resource is used. Our approach is to provide both kinds of propagation. We use two kinds of policies, "local" and "inheritable". A local policy applies only to the target on which it is imposed. An inheritable policy applies to a target and all its subtargets. This two policy approach adds some complexity, but something like this is needed in order to effectively handle policies involving multiple authorities.

Note that view restrictions on a policy may not completely hide all information about a policy, since users can attempt various operations and be allowed or denied. However, it can hide some information.

5.3 Local and Inheritable Policies

To support multiple control we introduce two different kinds of policies associated with a target object. The first is the local object policy for that particular target object. The second is an inheritable policy that applies to a target and all of its "subtargets". The full inherited policy on a given target then consists of policies from "higher" targets plus an inheritable policy for the target. The intent is that higher level authorizing agents, such as owners of higher level directories, can have some say on usage of their resource. Thus, the policy on a target object is the local policy plus all the inherited policies plus the target's own inheritable policy. The inherited policies are treated "by reference"; if the inheritable policy on a target changes, then that affects all the policies of the subtargets of that target.

The notion of a subtarget for policy inheritance depends on the specifics of the target class relationships. In the case of a file system the subtargets are the files and directories that are "contained_in" some directory (and all of the subobjects of the subdirectories). The inheritable policy for files is equivalent to an additional local policy, as there are no subtargets for files. To avoid confusion, inheritable policies for files should typically be avoided. (In the current demonstration implementation, inheritable policies for files are not allowed.)

5.4 Ease of Use

An important objective of this project is to make setting and modifying access policies easy for the user.

One way we are attempting to achieve this is by using a notion of user default policy (that can be tailored by the object creator) that defines the initial access policy for newly created objects. (Note that a user default policy is a meta-policy, because it consists of rules that determine which access policy should be associated with newly created objects.) Because the expression language is more expressive than standard access control mechanisms (for instance, the expression language has the ability to constrain access in terms of an organizational hierarchy), it is more likely that a default policy can be specified that will properly represent the desired security constraints.

A user may still need to modify his policy on some target object. Because of OCL's complexity, we have incorporated an extension of our language that hides some of the details. In particular, we have provided a generalization of an access control list (ACL) mechanism. However, if more complex rules are desired, the user will still be able to specify rules using OCL.

In our current design, each user has a default policy (composed of a set of policy specification rules) that define which access control policies are assigned to newly created targets. Both the user's access control policies and the user's policy specification rules can be modified by the user.

5.5 Policies on Policy Data

In addition to protecting access to targets, we need a way to protect access to security related information. This occurs in two ways. First, object security policies have view and modify rights that can be attached to them (Section 5.6.2). Secondly, the certificates could also be protected. (In the current implementation, there are no view protection rules on certificates or relationships.)

5.6 Policy Syntax

Policy statements are the rules used in determining whether an operation should be permitted. Access rules are evaluated to see if access is allowed. That evaluation must be based on supplied evidence. Trust rules describe the acceptability of the evidence.

We also introduce a syntax for defining the applicability of policies in new object creation (default policies). In a future version of the system, we may add an additional class of rules that controls the visibility of non-object information.

```
PolicyDecl := TrustPolicy | AccessPolicy | DefaultPolicy
```

All of the trust rules are grouped together as one policy.

5.6.1 Trust policies

```
TrustPolicy := "Trust" "Policy"
                  (trustRule   ("Response" name)?)*
               "End"
```

The trust rules are described in Section 5.8. The named response indicates how trust rule failures should be handled. We discuss the meaning of the response option in Section 5.9.

5.6.2 Access policies

Access policies contain rules governing access to a target, and constraints on how these access rules can be viewed and modified.

```
AccessPolicy := "Policy" <name> ("Response" name)?
                  (accessRule)*
                  ("modifyRight" OCLexpression)?
                  ("viewRight" OCLexpression)?
                  ("attribute" name securityaction OCLexpression)*
                "End"


securityaction := "read" | "write"
```

The main part of a policy is the set of access rules that it imposes. These are discussed in the next section. Note that the named response for an access policy indicates how failure of an access rule should be handled. We discuss the meaning of the response option in Section 5.9.

The `modifyRight` attribute indicates who can change the policy. In some sense, this clause defines who can control, or own, the object. It does not necessarily have to be the creator of that object. The `viewRight` attribute is a rule whose evaluation determines who can view the policy information.

The `modifyRight` and `viewRight` on new targets are initially setup using the initial policy as specified in the default policy specification. Users with modify rights can then change these rights. (In the current implementation of the demonstration, view and modify rights are application defined, and not under user control.)

Associated with a target may be certain (security) attributes. These are specified in the target specification class. Read and write restrictions can be imposed on these attributes. Some attributes should be set implicitly, e.g., date-of-creation. (There is currently no specification for how attributes are defined.)

5.6.3 Default policies

When a user creates a target, the user imposes a policy on that target. If the default policy specification is sufficiently well chosen, then the policy for the newly created object may require little or no alteration.

In order to keep the default policy specification simple, we define a mechanism that allows the selection of the appropriate initial policy for a given object. The default policy

specification is represented as a set of policy specification rules. The rules indicate which policy is appropriate for the newly created object.

Syntactically, the default policy specification consists of a list of use-when rules that specify a policy by name and an OCL expression. If the OCL expression is true when the target is created, then the named policy is imposed on the target.

Note that there are two default policy specifications; one specifies the local policy on a newly created target object, and the other specifies the inheritable policy.

These policies are specified with the following syntax.

```
DefaultPolicySpecification:= "Default" ("Local" | "Inheritable") name?
         ("use" policyname ("when" OCLexpression)*)*
End

policyname :=name
```

When evaluating the OCL expression, the system considers the "assign policy" request to have, as target, the newly created target object. Note that this request is different from the target creation request, since the original request is carried out on the directory in which the newly created target resides. (In the current implementation of the demonstration, the operation and requestor of this assign policy operation are the same as in the initial creation request.)

Ideally, the "use-when" OCL expressions should define a partition of the targets (for both local and inheritable policies). However, any ambiguity that is present is resolved by requiring that that the initial policy of an object is the first policy that makes one of the "use-when" OCL expressions evaluate to true. In the present design, if no use-when rule matches, then the user imposes no restriction on the target. An alternative would be to use an application defined initial policy if there was no match.

Note that policy names and not policies are used in the rules of a default policy specification. When a new object is created, the policy name is used to obtain the policy information. This information is copied and associated with the target object. At a later time, the owner of the target object can then modify the policy on that target object. (More complex schemes that allow the same policy to be shared by multiple targets are a possible design alternative, but are not part of the current design.)

When new users are added to the system, they are assigned local and inheritable default policy specifications (which define the policies that should be attached to newly created objects of that user). The default policy specifications are determined by the default policy initialization rules. These rules can be used to define the default policy specification based on characteristics of the new user.

```
DefaultPolicyInitialization :=  "Default"
```

```
                              ("Local" |"Inheritable") "Initialization"
            ("use"  Default_name ("when" OCLexpression)*)*
End

Default_name := name
```

In the current syntax, the default policy specification indicates whether it is for local or inheritable policies. Hence, the use of `Local` and `Inheritable` in the default policy initialization syntax is redundant, but useful for clarity.

When evaluating the OCL expression, there needs to be a way to refer to the new user that is being added to the system. The name `newUser` is used to refer to the object that represents the new user characteristics. The intent is that certificate information about a user is included in this object, prior to the choosing the default policy.

Ideally, the "use-when" OCL expressions in the default policy initialization rules should define a partition of the users (for both local and inheritable default policies). However, any ambiguity that is present is resolved by requiring that the first initialization rule whose expression evaluates to true is the one that applies. If no rule matches, then an error should be reported to the administrator of the system. One possible action is to abort the new actor creation operation.

In the current implementation of the demonstration, the default specification initialization is just a single hard-coded rule that applies to all users.

5.6.4 Special circumstances

A useful feature would be to allow different policies to be used in special situations, such as emergencies. Such policies would allow administrators, or other users, to override the access rules and then make the necessary changes to an object. Circumstances might include:

- Special operation status
- Priority override
- Repossession of resource

These types of requests might be handled as a different type of request, or special circumstances might be an attribute of a standard request.

Control for these special circumstances requires more care than standard access control. The results of evaluating the trust on certificates may need to produce more information than typical certificates in order to facilitate the evaluation of a special circumstances request.

Once this information is obtained, a possible solution is to bypass the standard access control mechanism and evaluate some OCL expression to check that the override is permitted for that requestor, operation, and target object.

It is possible to have multiple policies to allow different organizations to have different override constraints. However, a simpler solution may be to have one common policy, where the organizations can use different criteria for issuing override certificates.

This feature has not been implemented.

5.7 Access Rules

The main part of an access rule is an expression that may either be a boolean OCL expression or an access control list (ACL). An access rule that is a boolean OCL expression is a condition that an access request (together with the state of the system) may or may not satisfy. The expression must evaluate to true in order for access to be granted. Access control lists are not as expressive as OCL expressions, but are simpler to specify. An access control list consists of a list of pairs of expressions. The first part of the pair is a set of subjects (as in OCL a non-set element can be coerced into a singleton set) and the second part is a set of allowed actions. The rule is true when for some pair on the list, the requestor is in the set specified by the first part and the action requested is in the set specified by the second part. (Note that a set in an ACL pair does not have to be a set of constants; it could be, for instance, a set resulting from navigating a relationship.)

```
ACLexpression := ACL  (ACLpair)* EndACL
ACLpair := "(" OCLexpression "," OCLexpression ")"
```

An access rule can be expressed as just one rule or a set of subrules.
A rule that contains "subrules" is just the logical OR of the subrules. This syntax has been introduced to simplify the presentation in a user interface.

A variation on rules/subrules is also permitted. The language permits a "nested accretion/subtraction" syntax (similar to the year 1 report method [GG+97]). In this style, subrules are designated as either allow or deny. Each allow subrule permits requests which meet the subrule's condition. It is exactly the same as a "subrule". Each deny subrule removes permission if the subrule's condition is met. So, even if a previous subrule "grants" permission, a later deny subrule may remove it. A future allow subrule may then reestablish the permission. Thus, the ordering of the rules is part of the definition. This style of subrules has the advantage of letting users define permissions by incremental changes (additions and deletions). It has the disadvantage of being harder to visually determine when requests are permitted. The design permits, but does not require, this style. (This feature is not supported in the current implementation.)

For optimization reasons, a rule can contain regular subrules or allow/deny subrules, but not both. (If there are only subrules, then one can stop evaluating the rule when one of the subrules becomes true.)

```
accessRule := "Rule" (name)?
               (ruleexpression
              | ("SubRule" (name)? ruleexpression)+
              | (("Allow" (name)? ruleexpression)
                     |("Deny" (name)? ruleexpression))+


ruleexpression := OCLexpression  -- a general constraint
                  | ACLexpression  -- simpler form for some users
```

The expression of a rule or subrule can be an expression with an implication, and this can simplify evaluation.

For example,

```
((request.action=CREATE) implies "expression")
```

will succeed when `request.action` is not a `CREATE`, and hence "`expression`" will not need to be evaluated. (A rule optimizer could be used to speed up evaluations, but this was not an objective for this effort.)

Subrules are not implemented in the current version of the demonstration.

5.8 Trust Rules

The trust rules are used to determine the trustworthiness of information used by the policy rules. The most important of these is trustworthiness of the requestor's identity. These rules are constraints on whether a given piece of information is acceptable, or possibly, to what extent the information is acceptable.

The results of the trust evaluation can be used to construct an internal form of a certificate with a "trust level". The trust level can be used in the policy rules. The trust rules support different trust types for different types of certificates. If no trust type is given for a particular type of certificate then a default is used. If the application specification includes a class called `CertificateTrust`, then this class will be used as the default type of certificate trust for certificates. If no such class is specified, then integer values will be assumed.

The system access rules may use the results of the trust evaluation in determining access. For example, one of the application/system policy rules may insist that the requestor's certificate trust level be a "minimum" value.

The trust rules contain references to algorithms that will be used to perform the trust rule evaluation. Three predefined algorithms are available, which return values of 0 or 1 for failed and succeeded, respectively.

- no check (untrusted)
- signature validation
- hierarchical signature validation (check signature and signatures of higher-level CA's)

Other algorithms may be needed, and these are referred to by name. The implemented certificate class (or possibly the application) will contain the algorithm.

```
trustRule :=  "Rule" (name)?
          ( Certificate_type | "Default") trustmethod trusttype?

-- Certificate_type groups together a number of different elements
-- whose certificates are treated the same way

Certificate_type = (identity | relationship name | attribute name)+

trustmethod :=
        "checksignature"
      | "transchecksignature" relname --hierarchical check of signature
      | "nocheck"
      | String   -- name of a different algorithm to apply

trusttype := OCLexpression  -- type expression
```

Note that the current style of specification for trust rules does not have the same flexibility as that of the access rules – there is less specification and more hard coding of the trust evaluation method.

5.9 Audit and Failure Response Rules

Audit actions can be associated with access policies. The keyword `Response` followed by a name indicates the method to invoke for response handling. If no attribute is given, then the "default" method is invoked. If the rule has a name, then that name is passed as a parameter.

Since some rule evaluation may cause an error, it is useful to distinguish a rule evaluating to false from a rule failing to evaluate. An additional parameter is passed indicating the cause.

Some consideration should be given to the implementation of the response class. If a user is "denied because of insufficient rights", some information will be conveyed about the protection rules. A more detailed explanation might be considered an unacceptable security leak.

A different possible problem is that rule evaluation may be time-consuming or possibly even nonterminating. A time-out mechanism will be needed, if users are allowed to submit arbitrarily complex rules. There is no such mechanism in the current implementation.

5.10 Policy Properties

OCL could be used as an assertion language about the properties of policy rules. For example, one could use OCL to construct a property such as whether a certain class of users had some particular access right on some class of objects. These properties could then be checked to make sure the policy had the right characteristics. This is currently not supported.

There is ongoing work in supporting the analysis of properties specified in OCL [BG98]. (It is also possible that some of the policy analysis could be hard-coded.)

Note that since there is a specific operational mechanism associated with policy rule evaluation, the semantics for any policy will always be unambiguous (although certain rule evaluations could result in "errors"). However, a rule analysis might surface some unintended policy consequence.

5.11 Optimizations

Optimizations could be performed on a set of rules to minimize evaluation time. This is currently supported in only a limited fashion. Short-circuit evaluation is used for expression evaluation and for evaluating a set of subrules. (For the intended applications, accesses will not be frequent, thus this feature is not that important.)

6 System Classes

In this section, we describe the main classes of the language that are used in building the terms for evaluating access rules. The exact definition of these classes is dependent on the application. In this section, we describe these classes in a general way that could apply to most applications. A complete description of the classes for a particular example is described in Section 7.

Two key types of objects for describing the "system" for access control purposes are principals, which are authenticated identities, and targets. We will also introduce some other classes needed for access evaluation.

Although commonly used in security modeling, the term "principal" is generally not used in object modeling. Therefore, we use the object modeling term "actor". The UML Language Reference Manual defines an actor as "an idealization of an external person, process, or thing interacting with a system, subsystem, or class". In general, actors do not have to be human users in some role, but could be entities such as software applications. However, in our particular application, they are just humans. (Note that this use of the word actor is more constrained than that of Adage.)

A user will represent himself to the system using an identity certificate (or possibly some other authenticating information as well). A user who possesses multiple identity certificates will therefore be able to take on several identities from the system's point of view. A user will need to choose the appropriate identity with which to make a given

request. For example, a user may have an identity for performing unclassified work, and a separate identity for performing classified work.

Targets are the objects that a requested operation is supposed to act upon. We focus on targets that are files and directories.

6.1 Actors

Our scheme does not require a central authority that maintains identity information. Instead, many certifying authorities potentially have the capability of issuing identity certificates that contain cryptographically verifiable bindings between a user identity name and a public key. The access control system can accept and process these identity certificates (assigning an appropriate level of trust) and process requests from these users. For instance, a company that hires a new user can issue a new identity certificate for that user. If the access control system recognizes the validity of the identity certificate, then when the new user submits a request with the new certificate, the access control system can create a new internal actor representation with that certificate and then process the request. An identity certificate may either be issued by the CA of the organization of a user (if that organization has a CA), or by a third party that the organization signs. The access control system can have different levels of trust in a certificate depending on its contents. For instance, the access control system may have higher trust in a certificate with a certain CA's signature than with another's.

The access control system can maintain an internal representation of identity information or can compute it as needed. In either case, the identity information of a given actor will be (possibly conceptually) grouped together as an object. These objects are instances of a class called the "actor" class. Depending on the inheritance hierarchy, the actor class may be abstract, and these objects may be instances of a subclass. Attributes of the actor class will include characteristics such as the name, full namespace (sequence of names), expiration date, and trust characteristic of basic identity information. The system will maintain a connection between any internally stored information and the certificates that supply this information, in case the certificates are later revoked. When designed for military purposes, the certificates may contain security levels, and these should also be part of the internal representation of the individual. In a particular application, the system may also recognize signed certificates that convey additional attributes of an individual. These dynamic attributes are handled much like relationships (see Section 4.1.4).

Individuals can be grouped together using relationships. For example, one could introduce a relationship between actors representing some parameterized group. Then the groups can be built by supplying certificates that attest to all of the relationship instances. When an access rule is evaluated, it will be based on that evidence. (Note that the expression language allows other ways to specify sets of individuals.)

The design of the access control system is meant to support revocation requests. However, this is not implemented in the current demonstration.

Each actor has a default security policy that is used to assign a policy to newly created targets (see Section 5.6.3).

6.1.1 Handling multiple identity certificates

It is possible that a user could have more than one identity certificate. In this case, a mechanism to tell whether an actor A represents the same user as an actor B may be useful. One way is to form groups of related identities that designate the same user.

6.1.2 Default values

Not all attributes need to be specified when an actor is created. An object such as "default actor" could be part of the application specification and used to set defaults for unassigned attributes in new actors.

6.1.3 Sample actor class

Here is a sample actor class:

```
Class Actor
   full_name : String
   certificates : Sequence(Certificate)
   secrecy_level : Securitylevelrange
   integrity_level : Integritylevelrange
   expiration_date : Date
   trustlevel : Trust_type
End
```

6.1.4 Delegation

In the current version of the system, there is no specific support for uniformly expressing delegation policies (i.e., where one user is permitted to act on behalf of another user). However, one can express certain terms in access rules such as `(a in request.requestor.delegateof)`, where `delegateof` is a relationship between users.

6.2 Targets

A target is an object to which a user can request access. Like actors, targets can be refined in a number of ways. For the demonstration system of this project, targets are directories and files.

Associated with each target is a collection of security characteristics. It is this information (and not the actual target object) that is used in the security evaluation. This information is grouped together in the target specification class.

The syntax for introducing Target specification classes is different from other classes. This is because we may want to be able to specify attributes of target operations, and the fact that we do not need to apply those operations in an evaluation. See Section 4.1.1.2 for details about the syntax.

Here are sample target classes.

```
TargetSpecClass TargetSpec
   secrecy_level : Securitylevelrange      -- such as hierarchical level
   integrity_level : Integritylevelrange
   creator : Actor
End

TargetSpecClass File
   -- inherits "TargetSpec"  (implicit, as TargetSpec is
   --    automatically inherited)
   actions read,write,execute
End

TargetSpecClass Directory
   actions create,delete
End
```

Note that certain requests may refer (directly or indirectly) to an executable as well as a target object. This permits controlling the dynamic creation of new objects that are appropriate for the intended audience.

6.3 Actions

Actions represent the methods of the target object that are access controlled. In order to simplify the rules, we may also want to base access decisions on the properties of the methods of the target class (see Section 4.1.1.2). For example, an action can have a property that represents the kind of access that is involved, such as whether the access involves reading, writing, or both reading and writing. In particular, an operation like "get_creation_date" could then have an `accesstype` of `read`. (Adage has a different way of achieving this using a notion of generic_action.)

6.4 Requests

A "request" object is an abstract representation of a user request. Details of how the request's information was originally structured in the input, certificates, and signatures need not be part of the representation.

The primary type of request is a data access request, such as a read or write request. Here is a sample request class description.

```
Class AccessRequest
   originator : Actor
   object : Target    -- a "target class"
   operation : String -- possibly an enumerated type instead of String
   parameters: Sequence(String)
   -- two operations are "builtin" to simplify referring to requests
   -- that utilize one or two parameters
   Operation parameter1():String --first item on the parameter list
   Operation parameter2():String --second item on the parameter list
End
```

Another kind of request is one that sets or retrieves access control information on a target (such as modifying access rules).

```
Class InfoRequest
   originator : Actor
      object : Target
   operation : info_operation
   parameters: Sequence(String)
   Operation parameter1():String --first item on the parameter list
   Operation parameter2():String --second item on the parameter list
End
```

A possible set of info_operations for InfoRequests is: `VIEW_LOCALRULES`, `VIEW_INHERITABLERULES`, `MODIFY_LOCALRULES`, `MODIFY_INHERITABLERULES`, `ADD_CERTIFICATE`, `REVOKE_CERTIFICATE`, `VIEW_ATTRIBUTE`, `MODIFY_ATTRIBUTE`, `VIEW_METAPOLICY`, and `MODIFY_METAPOLICY`. The last two refer to viewing or changing the modify and view rights of the policy of some object. (In the current demonstration only the first four are implemented.)

6.5 Security Class

In some cases, it may be useful to have military security labels.

Rather than splitting the security label characteristics into different classes, as is currently done in Adage, we group them into one class. With this approach, the methods for comparing those levels will be contained within the security-level class. Here is an example where a security-level class contains a hierarchical level and a set of restrictive categories:

```
Type hierarchical_level
      enum{unclassified,confidential,secret,top_secret}
Type restrictive_categories Set(String)

Class security_level
   hier : hierarchical_level
   cat : restrictive_categories
   shared High : hierarchical_level
   shared Low:  hierarchical_level
   shared NONE,ALL: restrictive_categories
   Operator "<" (a:security_level)
   Operator "=" (a:security_level)
End
```

High is equivalent to the highest defined level and similarly for low, which in this case is top_secret and unclassified, respectively.

(The semantics of the operations would be in the Java executable classes that accompany the specification.)

6.6 Referring to the Objects of the Access Control System

The expressions of the access rules are allowed to contain names that represent objects. There are a few names that are reserved by the system.

- "`request`" refers to the object encapsulating the actor, target, and operation of a request.
- "`Master`" refers to the target object corresponding to the top-level directory
- "`newUser`" refers to a newly created user. This name is only used in default specification initialization rules that decide which default policy specification to assign to a newly created user.

Additional object names may be introduced by the application developer.

Other object names used in a rule are references to objects that are dynamically created by the system. If a name does not refer to an existing object (either because it was never created or was deleted) the value of the name is the `null` object.

6.7 External representations

Descriptions of the external format of the input requests to the system are hard-coded into the system. These include requests, certificates, and signature representations.

A somewhat more flexible version of the system could potentially make use of concrete specifications of these objects. This feature is currently not part of the language.

7 Example

7.1 Informal Explanation

In this example, we describe sample policies for a set of companies that want to collaborate on a government effort. We assume that the shared information and access control system will be situated on an unclassified government platform. The government can choose which companies can participate in the collaboration. The companies will authorize the individuals that can use this system. The companies will distinguish between project managers who have top-level control of their project and technical staff participating on this project.

The information will be organized as a directory structure. Each company will have its own subdirectory and each of those subdirectories will have subdirectories for different projects.

The individual role types are as follows:

- Master Authorizer– who has authority over the top level directory (designated with a dynamic attribute)
- New Project Authorizer – who has the ability to designate new projects and their managers (designated with a dynamic attribute)

- Project Manager – who has control over a subdirectory
  (designated implicitly, the owner of a project level directory)
- Technical Staff – who has control over a subdirectory of a project directory
  (designated by member_of_project)
- Credential Authority – who can assign relationships for some company
- Identity Authority – who can issue identity certificates
  (designated as part of the trust rules)
- Owner – users that control particular targets
  (designated as an attribute of the object)

7.2 Specification

Classes:

```
Class Actor
   name:String
   trustlevel:Integer
   company:Company
   -- subordinate_of uses the works_for relationship to see
   -- if there is a chain of works_for conditions between the actor
   -- object and the passed parameter
   Operation subordinate_of(person:Actor):Boolean
   -- the actorfor method converts the name of a user into an object
   -- if there is no match then a NULL value is returned.
   ClassOperation actorfor(name:String):Actor
   -- shortcuts for readability: check if requestor has authorization
   Operation hasMasterAuthorization():Boolean
   Operation isNewProjectAuthorizer():Boolean
End

Class Project
   name:String
End

Class Company
   name:String
End

Class Collaboration
   name:String
End

TargetSpecClass FileOrDirectory
   name:String -- name of directory
   owner:String -- initially set by creation
   Action read,delete
End

TargetSpecClass Directory
   Inherits FileOrDirectory

      Action createdir Property isCreate:Boolean is true
      Action createfile Property isCreate:Boolean is true
```

```
      -- possibly useful to directly refer to the company directory
      companydir:Directory
End

TargetSpecClass File
 Inherits FileOrDirectory
      Action write
End

-- Note in the demonstration
--   the actions are READ,DELETE,WRITE for Files and
--        READDIR,DELETEDIR,CREATEDIR,CREATEFILE for Directories
-- (Shared action names are not implemented)
```

Relationships:

Dynamic:

```
-- the ReportsTo relation indicates the staff hierarchy
-- It may be useful in configuring lower level directories
Relation ReportsTo
   Source Actor manages
   Destination Actor reportsto
End


-- the Member_of_project relation indicates that individual has
-- authority over part of a project directory
Relation Member_of_project
   Source Actor has_member
   Destination Project member
End

-- the Project_manager indicates that an individual manages some
-- project.
Relation Project_manager
   Source Actor manages
   Destination Project manager
End

-- the Corporate_participant relation indicates that a company can
-- participate in the collaboration
Relation Corporate_participant
   Source Company allows_corporation
   Destination Collaboration participates_in
End


-- Belongs_to indicates that an individual is part of some company
Relation Belongs_to
   Source Actor
   Destination Company belongs_to
End
```

Builtin:

```
-- Consists_of describes the files and directories belonging to
```

```
-- some directory

Relation Consists_of
   Source Directory parent multiplicity 0..1
   Destination FileOrDirectory children
End
```

Dynamic attributes:

```
-- NewProjectAuthorizer is the credential that a member of a company
-- must have to set up a new project and assign its project manager
Attribute NewProjectAuthorizer
   Source Actor
   Destination Boolean
   Default false
End

-- Master Authorization is the privilege that is necessary to do any
-- operation on a top level directory, other then setting up a new
-- project

Attribute MasterAuthorization
   Source Actor
   Destination Boolean
   Default false
End
```

Context:

```
-- The request object is a representation of individual requests to
-- the server.   It can include a parameter that is used to specify
-- names of new files or directories

Class Request
   requestor : Fullname
   target : Fullname
   operation : Name
   parameterlist : Sequence(String)
End

-- Note that in the current implementation, Fullnames are implemented
-- as names

-- Object values that are accessible as part of the context:
--   Actors, files
-- The actions are already defined as part of the target specification
-- classes.
```

Trust Rules:

To establish trust, proper signatures are required from an authorized company agent for both identity and relationship information. In this example, the trust attribute is 0 or 1 if not a member of an authorized company and >1 otherwise. Certificates with trust level 0 or 1 should simply be discarded as part of the trust evaluation mechanism, and so this value is really not needed as part of a certificate. However, for the purposes of showing how this could be used in access decisions, we refer to the value in our example rules.

We call this algorithm for establishing trust "invalid_unauthorized_or_authorized". It is expressed as:

```
Rule trust_evaluation Default
   "invalid_unauthorized_or_authorized" integer
```

Default Policy Specification

```
-- Default policy specifications are used to set the initial policies
-- of a newly created object.
-- This specification describes the default policies initially
-- provided by the system for new users.  (In the current demonstration
-- all new users must start with the same default policy.  There is no
-- default policy initialization specification.)

Default Local
   use localmaster when (request.target=Master)
   use localsecondlevel when (request.target.parent=Master)
   use lowerlevel when true -- in any other case
End

Default Inheritable
   use localmaster when (request.target=Master)
   use inheritablesecondlevel when (request.target.parent=Master)
   use inheritablelowerlevel when true -- in any other case
End
```

Policies for Master

This is the local policy of the top-level directory. An actor with new project authorization privilege can create or delete a directory. All other actions require master authorization.

```
Policy Local localmaster
Rule
   if (request.requestor.hasMasterAuthorization())
     then true
   else if request.requestor.isNewProjectAuthorizer()
      then (request.operation in Set{createdir,deletedir} )
            and (request.parameter1() = request.requestor.company.name)
   else
      false
   endif endif
End
```

```
Policy Inheritable inheritablemaster
Rule
-- note trustlevel is assigned to 0 or 1 if not a member of a currently
--authorized company or improper certificate.  Trust level>1 if
--certificate OK and authorized company
  request.requestor.trustlevel>1
End
```

Policies for 2ndLevel directories

```
Policy Local localsecondlevel
-- to delete or create a directory at this level must have
--  company authorization or be the manager of the project
--  Others can read (note that the master policy requires that
--   they be from one of the collaborating companies)
Rule
   if   ((request.operation=createdir) or
(request.operation=deletedir))          then
         (request.requestor.manages.name = request.parameter1())
   else
      (request.operation=read)
   endif
End


Policy Inheritable inheritablesecondlevel
Rule
(not (request.target.company in request.requestor.belongs_to)) implies
     (request.operation=read)
End
```

Policies for lower level directories

```
Policy Local locallowerlevel
--Default Inheritable policy for most newly created objects requires --
-- that the requestor is the owner, or access is limited to
--read (for file or directory)
Rule
   (request.requestor <> request.target.owner) implies
      (request.operation=read)
End

Policy Inheritable inheritablelowerlevel
   -- no inheritable constraints
End
```

8 Description of Demonstration Software

In the preceding sections, we have described a language for specifying access control policies. This section presents a short description of the software that was built to demonstrate the access control methodology. We have built a prototype information sharing application that controls access to a repository of target files and directories. The

demonstration uses certificates for user identification, and simulations of certificates for other information such as relationships. It allows the user to create files and directories with specified user default policies, and to view and modify rules on target objects. Details of the installation and operation of this demonstration are in the Software User Manual deliverable [FR99].

The prototype is intended only to demonstrate the usability and flexibility of the access control mechanism, and does not provide the full security infrastructure needed for an information sharing product.

8.1 Overview

The top-level architecture consists of an information server and a user interface for interacting with the information server. The information server includes an access control engine for evaluating the security rules and an object manager for handling files and directories. User interfaces include support for retrieving and storing data, as well as modifying access policies on targets and default policies of users.

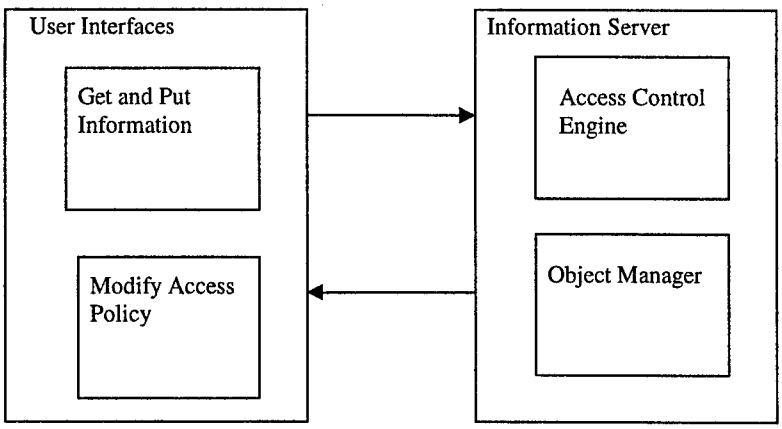

Figure 1: Components of Information Sharing Application

Important supporting components are certificate generation and evaluation. Our access control demonstration is a Web-based mechanism for distributed information sharing. The information resides on a Web server and permits users to access such information using a Web browser such as Netscape. Users can submit requests to view or modify information or rules, and access is protected by the policy-based access control mechanism.

The user can

1. Navigate the directories containing shared information
2. Read, write, create, and delete files
3. Create new subdirectories
4. View and modify rules on a target file or directory
5. View and modify the default policies that he assigns to newly created targets

There is also a (non-access controlled) interface to view and modify relationship; this interface is provided as a way to simulate changes in relationship information.

8.2 Client GUI web pages

The client interacts with the Web server via HTML web pages, augmented by JavaScript functions and served dynamically from the Web server. The user interface presents an integrated view of the target hierarchy within a Web-browser environment. A directory reader page displays the choices available from a given directory: there is a list of subdirectories and files, and choices of operations to perform on each. All operations are access-controlled by a Java servlet in the server intercepting the HTTP request running on the server side. When a client browser initiates a session with the server, it sends along a certificate to authenticate the client's identity. This identity is then used to process access requests made during this session.

The user can choose a file or directory (including the current directory) on which to perform an operation, and then pick an operation to perform. Currently supported operations on directories are to read the directory, create a file within the directory, create a directory within the directory, delete a directory within the directory, and view or modify policies on that directory. Operations supported on files are to read a file, write to the file, view or modify policies on the file, or delete the file. Once the user has selected the target and operation, the user presses a submit button to send the request to the server for evaluation and execution.

8.2.1 Reading, writing, and creating files

Requests to read, write, or create files are all access controlled; successful requests return pages to perform the requested functions.

8.2.2 Viewing and modifying rules on a target

A user can request to view or modify rules on a target. The user can then choose which of the policies imposed on the target to view or modify. These policies are the local and inheritable policies of the target itself, along with the policies inherited from the directories containing the target. Whether this request is successful depends on the view and modify rights of the policy on which the request is made. The current implementation supports viewing and modifying complex rules (without subrules or allow/deny rules) and ACL rules.

8.2.3 Viewing and modifying relationships

This page allows a user to view and modify relationships of the system. The page has three lists, the first of which contains the relationships of the system. When one of them is selected, the second list displays the names of the start objects of the selected relationship. When one of the start objects is chosen, then the third list displays the end objects related to the selected start object. The present page also simulates the action of adding and revoking relationship instances based on relationship certificates.

The interface does not currently display or modify dynamic attributes.

PART III: Java Enhancements and Tools

9 PolyJ

Java [Sun95a] is a type-safe, object-oriented programming language that is used increasingly widely for mobile applications, e.g., for so-called *applets* that can be used to provide active content on web pages. Java achieves code mobility by using a machine-independent target architecture, the Java Virtual Machine (JVM) [LY96]. Java provides some degree of safety for mobile applications, in part because Java bytecodes can be verified statically, preventing violations of type safety that might access private information. Java also enforces stricter run-time checking than languages like C and C++. Because of the widespread interest in Java, its similarity to C and C++, and its industrial support, many organizations are making Java their language of choice.

One problem with Java, as currently defined, is that certain coding errors are detected only when a program is run, not when it is compiled. This problem causes Java programs to be less reliable than could be achieved with additional compile-time checking; that is, it brings coding errors to the attention of the people who use code rather than the people who developed that code. One particular failing in Java, which unduly limits the amount of compile-time checking that can be done, lies in its lack of support for generic code. In Java, it is possible to define a new type, such as a set of integers, but it is not possible to define a generic abstraction for sets, in which the elements of a particular set are homogeneous, but the element type can differ from one set to another. Current Java programs must adapt to the lack of genericity by using run-time type discrimination, which is error-prone, awkward for the programmer, and relatively expensive.

To address this problem, we have extended Java with parametric polymorphism, a mechanism for writing generic interfaces and implementations. The resulting language, called PolyJ [MBL97], is supported by a portable compiler, which translates PolyJ programs into standard Java programs, which can be compiled by any standard Java compiler.

An explicit goal of our work was to be very conservative. We extended Java by adapting an existing, working mechanism with as few changes as possible. We supported the Java philosophy of providing separate compilation with complete intermodule type checking, which also seemed important for pragmatic reasons. We used the Theta mechanism [LCD+94, DGLM95] as the basis of our language design, because Theta has important similarities to Java. Like Java, it uses declared rather than structural subtyping, and it supports complete intermodule type checking. We rejected the C++ template mechanism [Sto87] and the Modula-3 [Nel91] generic module mechanism because they require that a generic module must be type checked separately for every distinct use. Furthermore, the most natural implementation of these mechanisms duplicates the code for different actual parameters, even when the code is almost identical.

9.1 Illustration of Unreliability in Java

The following is an example of Java code that compiles without any error reports, but that produces an error at run-time.

```
import java.util.Enumeration;
import java.util.Vector;

public class sample {
public static void main(String[] args) {
   Vector v = new Vector();
   v.addElement("abc");
   v.addElement(v);
   for (Enumeration e = v.elements(); e.hasMoreElements(); )
      System.out.println(v.nextElement());
   }
}
```

When run, the compiled code produces one line of output followed by 6,325 lines of error messages:

```
abc
java.lang.StackOverflowError
   at java.lang.StringBuffer.append(StringBuffer.java)
   at java.util.Vector.toString(Vector.java)
   at java.util.Vector.toString(Vector.java)
   ...
   at java.util.Vector.toString(Vector.java)
   at java.io.PrintStream.print(PrintStream.java)
   at java.io.PrintStream.println(PrintStream.java)
   at sample.main(sample.java:9)
```

Clearly, such behavior is unacceptable. Developers should be able to detect and correct such defects before shipping applications to users. Users should be able to depend on receiving reliable code and should not be subjected to strange run-time failures.

The underlying problem with this code is that Java provides no good way for the programmer to express the intent that the vector `v` contains only strings and not other types of objects. Hence the statement

```
v.addElement(v);
```

is not detected as an error during compilation, and the statement

```
System.out.println(v.nextElement());
```

produces an error at run-time when it attempts to print the value of the second element in `v`. Even if the programmer rewrites the second statement as

```
System.out.println((String)v.nextElement());
```

to cast the elements of `v` to strings before printing them, the program still compiles without any errors being reported and produces the following run-time error:

```
abc
java.lang.ClassCastException:
   at sample.main(sample.java:9)
```

Although this error message may be less objectionable than before, the code is no more reliable, and the user is not likely to be any happier with the result.

9.2 Improved Reliability in PolyJ

PolyJ enables programmers to annotate their code so that errors of the kind just illustrated can be detected at run-time. For example, a PolyJ programmer can rewrite the above code as follows to indicate that all elements in the vector `v` are strings.

```
import polyj.util.Enumeration;
import polyj.util.Vector;

public class sample {
public static void main(String[] args) {
   Vector[String] v = new Vector[String]();
   v.addElement("abc");
   v.addElement(v);
   for (Enumeration[String] e = v.elements(); e.hasMoreElements(); )
      System.out.println(v.nextElement());
   }
}
```

Here, `Vector[String]` is an instantiation of a parameterized type `Vector[T]`, and `Enumeration[String]` is an instantiation of a parameterized interface `Enumeration[T]`. With the additional information provided by type parameters, the PolyJ compiler is able to detect the error in the program and report

```
sample.pj:7: No matching method:
addElement(polyj.util.Vector[java.lang.String]) found on class
polyj.util.Vector[java.lang.String]
```

to the programmer of `sample`, who can repair the error before the user of `sample` ever runs the code.

The source code that defines the PolyJ `Vector` abstraction is obtained by changing the first few lines of the Java `Vector` abstraction from

```
public class Vector implements Cloneable {
   protected Object[] elementData;
   protected int elementCount;
```

to

```
public class Vector[E]
where E { boolean equals(E e); String toString(); }
  implements Cloneable {
   protected E[] elementData;
   protected int elementCount;
```

and then systematically replacing all other occurrences of `Object` in the source code by `E` and all occurrences of `Enumeration` in the source code by `Enumeration[E]`. The identifier `E` is a type parameter, which can be instantiated by types such as `String` that have methods satisfying the `where` clause in the definition of `Vector`.

The information in the `where` clause serves to isolate the implementation of a parameterized definition from its uses (i.e., instantiations). Thus, the correctness of an instantiation of `Vector` can be checked without having access to a class that implements `Vector`; in fact, there could be many such classes. Within such a class, the only operations of the parameter type that may be used are the methods and constructors listed in the `where` clause for that parameter. Furthermore, the class must use the routines in accordance with the signature information given in the `where` clause. The compiler enforces these restrictions when it compiles the class; the checking is done just once, no matter how many times the class is instantiated.

A legal instantiation sets up a binding between a method or constructor of the actual type, and the corresponding `where-routine`, the code actually called at run-time. Since an instantiation is legal only if it provides the needed routines, we can be sure they will be available to the code when it runs. Thus, the `where` clauses permit separate compilation of parameterized implementations and their uses, which is impossible in C++ and Modula 3.

9.3 Implementation Strategy

We implemented PolyJ using a source-to-source translation, which transforms PolyJ programs into legal Java programs that can be compiled by any standard Java compiler. Although more efficient run-time code could be produced by adding new byte codes to the Java Virtual Machine, we chose not to do so, thereby enabling PolyJ programs to be run on any Java Virtual Machine.

The compiler is built as an extension to the `guavac` compiler developed by David Engberg [Eng96]. It is written in C++, which also means that it runs efficiently on virtually all platforms.

There are two reasonable ways to implement parameterized types in Java without extending the virtual machine. The most obvious implementation is to treat each instantiation of a parameterized class or interface as producing a separate class or interface. Each instantiation of a parameterized class would have its own ".class" file that must be separately loaded into the interpreter and verified for correctness. In essence, the parameterized code is recompiled for each distinct set of parameters. This technique is similar to the template implementation used by most C++ compilers, which leads to substantial code blowup. It differs from the C++ approach in that the `where` clauses guarantee successful recompilation.

We employed an alternative strategy, which produces code that is generic across all instantiations: the compiler generates bytecodes for parameterized classes as though all parameters are of class `Object`. When compiling code that uses a parameterized class, the compiler generates run-time casts as appropriate. Because the compiler has type checked the code, all the run-time casts necessarily succeed, but the performance is the same as for old-style Java code that manipulates variables of type `Object` and performs explicit casts.

In this scheme, invocation of `where-routines` is complicated. Each object of a parameterized class contains a pointer to a separate object that bundles up the appropriate `where-routines` for the instantiation, presenting them as methods. The compiler translates a `where-routine` invocation to an invocation of the corresponding method of this `where-routine` object. The `where-routine` object is installed in the object of a parameterized class by passing it as a hidden, extra argument to class constructors. The advantage of this technique over the previous one is that only the code of these `where-routine` objects is duplicated for each instantiation; most of the code of a parameterized class is shared for all instantiations.

9.4 Comparison with Other Approaches

PolyJ differs from other approaches to providing parametric polymorphism in several respects. Unlike some approaches, it uses the constraint mechanism of `where` clauses, which is important because it provides flexibility when composing a program. It also allows basic types like `int` to be used as parameter types. Unlike some other approaches, instantiation types are first-class types that may be used wherever a type may be used, particularly, in run-time casts and `instanceof`.

PolyJ uses `where` clauses rather than subtype constraints as is done in several other approaches (e.g., Pizza [ORW97] and GJ [BOSW98]) to providing parametric parameterization. The problem with subtype relationships is that Java, unlike most of the object-oriented languages with parametric polymorphism, requires explicit declaration of subtype relationships. Thus, in order to create an instantiation type such as `Set[Node]`, there must be an explicitly declared subtype relationship between `Node` and a special parameterized constraint interface.

There are two problems with this requirement. First, Java is supposed to support development with extensive libraries and separate compilation. It is both limiting and contrary to the spirit of Java to require that a programmer wishing to use `Set[Node]` have access to the source code for `Node`. Furthermore, there would be an explosion of parameterized constraint interfaces, and many classes will have to declare that they implement a long list of them. The explosion would be particularly bad because Java supports overloading, and different generic classes would require different versions of the overloaded method. These different versions would lead to different constraint interfaces that differ only in how they are overloaded.

10 Enhanced javadoc Utility

We produced a prototype of Percolator, a tool that generates html documentation files for Java programs. Percolator is similar in function to the Sun javadoc utility. Like javadoc, Percolator extracts documentation from stylized comments embedded in source code. Unlike javadoc, it checks the extracted documentation for consistency with the source code. Checks currently supported by Percolator include:

- Parameter and exception consistency: any parameter or exception documented in an `@param` or `@exception` section must correspond to a parameter or exception in the method definition.

- Returns consistency: the documentation for a method that does not return a value may not have an `@returns` section.

- Completeness: all parameters, exceptions, and returned values must be documented in an `@param`, `@exception`, or `@returns` section (unless the `-mustdocument` command-line option is used).

Percolator also recognizes the following additional documentation sections, which are motivated by the design of Larch interface specification languages [GH+92].

- `@requires`: documents requirements that must be satisfied by the caller before invoking a method.

- `@effects`: documents the effects of executing method.

- `@modifies`: lists the objects whose values may be changed by invoking the method.

The following additional annotations and checks would be useful in Percolator, but have not been implemented.

- support and checking for PolyJ parameterized types,

- "maybe" type annotations and checks to prevent run-time errors that occur when dereferencing a null pointer, and

- preconditions for native methods and checks to provide lightweight security validation.

PART IV: Enforcing Safety Properties

11 Enforcing Safety Properties

People often want to run programs without allowing them total control over their system. Users should not have to worry about buggy or malicious programs disrupting other programs, corrupting files on their disk, or compromising their privacy.

Modern operating systems provide some support by protecting process address spaces and setting access permissions on resources, but are limited by how much information they have about the acceptable behavior for a specific application. For example, behavior that is normal for a network backup utility would be considered suspicious for a web applet. Although a general-purpose operating system cannot be expected to know the limits on acceptable behavior for an arbitrary program, an application writer can. Even a naive end-user has a reasonable notion of what different types of programs should not be expected to do.

We have developed a system, called Naccio [Evans99, ET99], that enables system administrators or end-users to impose general safety policies on the programs they intend to run. With the aid of Naccio, administrators or users can specify limits on the acceptable behaviors of programs as safety policies, which consist of collections of safety properties. Naccio tools transform programs to produce trusted programs that satisfy specified safety policies.

Examples of the kinds of safety properties that Naccio is designed to support include:

- Restricting what files or directories may be read and written.
- Requiring that any temporary files created by the program be removed before execution terminates.
- Placing a limit on the total amount of disk space that may be used by files created by the program.
- Prohibiting the application from communicating with certain IP addresses.
- Prohibiting the application from communicating with other hosts after sensitive files have been read.
- Limiting the fraction of available network bandwidth the application may consume during any five-second period.

11.1 Overview of Naccio

Suppose, for example, one wishes to enforce a safety policy that places a limit on the total number of bytes applications may write to files. To do this, the system needs to maintain a state variable that keeps track of the total number of bytes written so far. Furthermore, it needs to check before every operation that writes to a file that the limit will not be exceeded. One way to enforce a safety property like this would be to rewrite the platform libraries to maintain the necessary state and do the required checking. This would require access to the source code of the platform libraries, and we would need to rewrite them each time we wanted to enforce a different safety policy.

Instead, we could write wrapper functions that perform the necessary checks and then call the original system function. To enforce the policy, we would modify the target program to call the wrapper functions instead of calling the protected system calls directly. Though wrappers are a reasonable implementation technique, they are not appropriate for describing safety policies, since creating or understanding them requires intimate knowledge of the underlying system. To implement the write limit policy, the author of the safety policy would need to identify and understand every system function that may write to a file. For even a simple platform like the Java API, this involves knowing about more than a dozen different methods. Changing the safety policy would require editing the wrappers, and there would be no way to reuse a safety policy across multiple platforms.

The Naccio solution is to express safety policies at a more abstract level and to provide tools that generate and use the appropriate wrappers to enforce a safety policy on a particular platform. We express safety policies as constraints on resources and characterize system calls by how they affect those resources.

Figure 2 shows the Naccio system architecture, which incorporates a *policy generator* and an *application transformer*. A policy author runs the policy generator to produce what the application transformer uses to enforce the policy on a particular program. Since policy generation is a relatively infrequent task, Naccio trades off execution time for the policy generator to make application transformation fast and to reduce the run-time overhead required to enforce the policy. Once a policy has been generated, it can be reused to constrain the operation of multiple applications.

The policy generator uses the following inputs to produce a *policy-enforcing platform library*, a version of the platform library that includes the wrappers necessary to enforce the policy. It also produces a *policy description file* that contains the transformation rules required to invoke the wrappers and enforce the policy.

- *Resource descriptions* (abstract descriptions of system resources). Naccio provides a set of standard resource descriptions. Policy authors can alter this set, provided they make corresponding alterations in the platform interface. However, a wide variety of safety policies can be expressed using Naccio's standard resource descriptions.

- The *safety policy* (a description of the constraints to be enforced on the manipulation of abstract resources). Policy authors (i.e., system administrators or individual users) write these in terms of the abstract resource descriptions.

- The *platform interface* (a description of the particular system platform that describes the effect of system calls on abstract resources). As for resource descriptions, Naccio provides standard platform interfaces (e.g., for Java API classes and Win32 Dynamic Link Libraries). Policy authors do not need to know the contents of the platform interface.

- The *platform library* (the unaltered platform library, for example, the Java API classes or Win32 Dynamic Link Libraries).

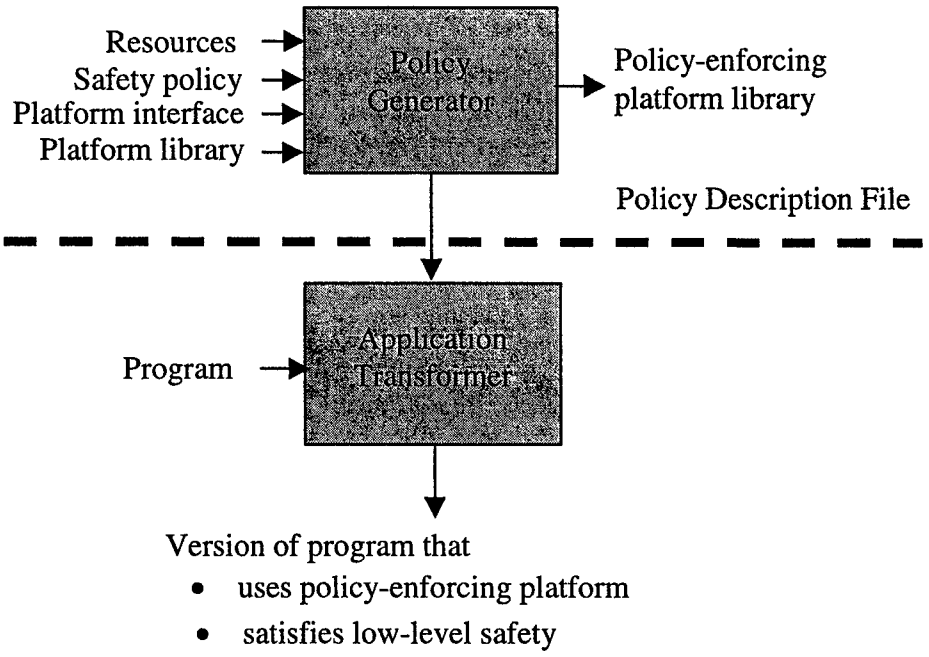

Figure 2: Naccio Architecture

The *application transformer* is run when a user or system administrator elects to enforce a particular policy on an application. The transformer applies the transformations in a policy description file to the target program to produce a version of the program that is guaranteed to satisfy the safety policy. It does this by replacing system calls in the program with calls of the policy-enforcing library. It also ensures that the resulting program satisfies the low-level code safety properties necessary to prevent malicious programs from circumventing the high-level code safety mechanisms. Standard

techniques for low-level code safety, such as bytecode verification or software fault isolation, are used for this purpose.

The application transformer needs to be run only once for each program and safety policy. Afterwards, the resulting program can be executed normally.

In the following sections, we illustrate the form of the inputs provided to the policy generator. The operation of the policy generator and the application transformer are described in [Evans99], [ET99], and [Twy99].

11.2 Describing Resources

Resource descriptions provide a way to identify resources and the ways in which they are manipulated. Examples of resources include a file system, a network connection, a system property, and a thread. Resource descriptions are platform independent, but may be used to describe platform specific resources such as the Windows registry.

Resources in Naccio are described by listing their operations and observers. Resource operations have no implementation; they are merely hooks for use in describing safety policies. The meaning of a resource operation is indicated by its associated documentation. The essential promise is that a transformed program will invoke the related resource operation with the correct arguments whenever the documented event would occur. It is up to the policy generator and platform interface to ensure that this is the case.

Following is an excerpt from an `RFileSystem` resource description that describes a resource corresponding to the file system. The **global** modifier indicates that only one `RFileSystem` instance exists for an execution. Resources declared without a **global** modifier are associated with a particular run-time object. Most of the `RFileSystem` operations take an `RFile` parameter, a resource object that identifies a particular file.

global resource `RFileSystem`
  operations
    `initialize ()`
      "Called when execution starts."
    `terminate ()`
      "Called just before execution ends."

    `openRead (file: RFile)`
      "Called before file is opened for reading."
    `openCreate (file: RFile)`
      "Called before a new file is created for writing."
    `openOverwrite (file: RFile)`
      "Called before an existing file is opened for writing."
    `openAppend (file: RFile)`
      "Called before an existing file is opened for appending."
    `close (file: RFile)`
      "Called before file is closed."

```
preWrite (file: RFile, n: int)
```
"Called before up to n bytes are written to file."
```
postWrite (file: RFile, n: int)
```
"Called after exactly n bytes were written to file."
```
preRead (file: RFile, n: int)
```
"Called before up to n bytes are read from file."
```
postRead (file: RFile, n: int)
```
"Called after exactly n bytes were read from file."

```
delete (file: RFile)
```
"Called before file is deleted."

```
observeExists (file: RFile)
```
"Called before revealing if file exists."
```
observeWriteable (file: RFile)
```
"Called before revealing if file is writeable."

Resource manipulations may be split into more than one resource operation. For example reading a file is divided into `preRead` and `postRead` operations. This division allows more precise safety policies to be expressed, since some information, in this case the exact number of bytes read, may not be available until after the platform call is made. Pre-operations allow necessary safety checks to be performed before the action takes place, while post-operations can be used to maintain state and perform additional checks after the action has been completed and more information is available.

Resource descriptions also define observers, functions that reveal some aspect of the state of a resource, but do not modify that state. Naccio provides observers so that safety properties can determine information about a resource in a platform independent way.

11.3 Expressing Safety Policies

The class of safety policies that can be expressed using Naccio is limited by the resource operations defined in the resource descriptions. Safety policies can constrain how these resources may be manipulated, but only in terms of information available through resources. For example, Naccio cannot express any liveness properties (i.e., properties that constrain what must happen at some time in the future).

A safety policy is described by listing safety properties and their parameters. Safety policies can be combined to define a new policy. The simplest way to combine safety policies is to create a policy that includes all safety properties of both policies. Other combination mechanisms designed to support easily modifying existing policies by strengthening or weakening particular constraints are currently under investigation.

`LimitWrite` is a sample safety policy that combines two safety properties. The `NoOverwrite` property disallows replacing or altering the contents of any existing file, and the `LimitBytesWritten(1000000)` property places a million-byte limit on the amount of data that may be written to files. These safety properties are defined in a

standard properties library. By itself, `LimitWrite` would not be a good policy to use for an untrusted application, because it constrains neither what files the application reads nor how the application uses the network. In practice, this policy would be combined with policies that place constraints on other resources.

```
policy LimitWrite
   NoOverwrite, LimitBytesWritten (1000000)

property NoOverwrite
   check RFileSystem.openWrite (file: RFile),
         RFileSystem.openAppend (file: RFile),
         RFileSystem.delete (file: RFile)
      violation ("Attempt to overwrite file: " + file.getName ());

property LimitBytesWritten (limit: int)
   requires TrackTotalBytesWritten;
   check RFileSystem.preWrite (file: RFile, n: int)
      if (bytes_written+ n > limit)
         violation ("Attempt to write more than " + limit +
                    " bytes.  Already written " + bytes_written +
                    " bytes, writing " + n + " more to " +
                    file.getName () + ".");

stateblock TrackTotalBytesWritten
    addstate RFileSystem.bytes_written: int = 0;
    postcode RFileSystem.postWrite (file: RFile, n: int)
      bytes_written += n;
```

A safety property consists of one or more **check** clauses that identify resource operations and provide action code for enforcing the property. For example, the **check** clause of the `NoOverwrite` property identifies the two `RFileSystem` resource operations associated with opening an existing file for writing (`openWrite` and `openAppend`) and the operation associated with deleting a file (`delete`). The action code simply invokes the **violation** command, which will produce a dialog box that alerts the user to the safety violation and provides an option to terminate the program.

The `LimitBytesWritten` property illustrates how a more complicated safety property is defined. It takes an integer `limit` parameter, and constrains the total number of bytes that may be written by the application to the value of that parameter. When `LimitBytesWritten` is instantiated in a safety policy, `limit` is given a value. To implement the write limit, we need to keep track of how many bytes are written. This is done by the `TrackTotalBytesWritten` state maintainer that is included by the **requires** clause. It adds a state variable to the `RFileSystem` resource, and provides a postcode action to the `RFileSystem.postWrite` operation. The body of this postcode action will be executed after all checking code associated with the `write` operation.

Hence, when the `LimitBytesWritten` property check action compares the value of `bytes_written` to `limit`, the value of `bytes_written` will not have been incremented before the comparison. It is advisable to keep the state maintenance and property checking code separate, since many safety properties may require the same state.

11.4 Describing Platform Interfaces

In order to enforce a safety policy, the appropriate resource operations must be called when the corresponding resource is manipulated. The platform interface describes how system calls affect resources.

For each platform, we must determine an appropriate level for the platform interface. That is, we must decide which library calls are described by the platform interface, and which will be considered part of the application and transformed by the application transformer. The level of the platform interface limits the resource manipulations that can be identified and, hence, the class of safety policies that can be expressed. For example, if we place the platform interface at the level of system calls, we cannot express safety policies that constrain lower-level resources such as memory or processor usage.

For Naccio/JavaVM, we are limited by our ability to deal easily with code for native methods. At a minimum, this means that the platform interface must describe all Java API native methods that affect resources. We could stop there and simply consider the rest of the Java API as part of the application that needs to be processed by the application transformer. When a library native method is called, the transformer could replace the call with a call to a wrapped version of the method that performs the necessary safety checking.

This would be unsatisfactory, however, since it allows for no distinctions regarding how the native method is called. For example, the AWT method that loads a font would call the same wrapper file open method as user code that opens a `FileInputStream`. To handle checking correctly, we would need to resort to using run-time mechanisms (e.g., stack introspection [WF98]) to identify and distinguish trusted system code. Instead, it seems clear that the platform interface for the JavaVM platform should describe all Java API methods in terms of how they affect resources. Internal API calls within API methods need not incur additional safety checks since the platform interface describes all relevant resource manipulations at the level of the API method. This eliminates the need to make any run-time distinctions between unprivileged code and privileged system code.

For each Java API method or constructor that affects a system resource, the platform interface must provide a wrapper that invokes the appropriate resource operations. In the following excerpt from the platform interface for the `java.io.FileOutputStream` class, the **requiredif** `RFile, RFileSystem` statement indicates that constructors and methods in this class need be wrapped only if the `RFile` and `RFileSystem` resources require checking. The `RFileMap` helper class keeps a mapping between Java files and the corresponding `RFile` objects. The `rfile` state variable keeps track of the `RFile` object associated with a `FileOutputStream`.

```
wrapper java.io.FileOutputStream requiredif RFile, RFileSystem {
   requires java.io.RFileMap;

   state RFile rfile;

   helper static RFile doOpen (java.io.File file) {
      RFile trfile;
      trfile = RFileMap.lookupAdd (file);
      if (file.exists ()) RFileSystem.openOverwrite (trfile);
      else RFileSystem.openCreate (trfile);
      return trfile;
   }

   wrapper FileOutputStream (java.io.File file) {
      RFile trfile;
      trfile = doOpen (file);
      #;
      rfile = trfile;
    }

   wrapper void write (int b) {
      if (rfile != null) RFileSystem.preWrite (rfile, 1);
      #;
      if (rfile != null) RFileSystem.postWrite (rfile, 1);
   }

}
```

Consider the wrapper for the `write` method. If the `rfile` object associated with this `FileOutputStream` represents a file, the wrapper calls the `RFileSystem.preWrite` and `RFileSystem.postWrite` resource operations. If the safety policy constrains the `write` operation, the relevant checks will be done in these operations. If there is a safety violation, it will be detected in the `preWrite` resource operation, and the user will have the option to terminate execution before the `FileOutputStream write` method is called. The # symbol indicates the point at which the original `FileOutputStream.write` method will be invoked.

PART V: Information Flow

12 Information Flow

The growing use of mobile code in downloaded applications and servlets has resulted in an increased interest in robust mechanisms for ensuring privacy and secrecy. A crucial problem is that information must be shared with downloaded code without allowing that code to leak the information. Information flow control is intended to address these privacy and secrecy concerns. However, most information flow models are too restrictive to be widely used. During this project, we have developed a promising new model, the IFlow decentralized label model for information flow [ML97, ML98, Myers99a, Myers99b].

Our goal is to allow a node to share information with a downloaded applet or uploaded servlet, yet prevent the mobile code from leaking the information. Additionally, it should be possible to prevent the applet or servlet from leaking private information to other programs running on the same node. Our approach is to check information flow by a straightforward static analysis of annotated program code.

The IFlow model makes a good basis for information flow control because it improves on earlier models in several ways:

- IFlow allows individual principals to attach flow policies to pieces of data. The flow policies of all principals are reflected in the *label* of the data, and the system guarantees that all policies are obeyed simultaneously. Therefore, IFlow works even when principals do not trust one another.

- IFlow allows individual principals to declassify data by modifying their own flow policies. Principals cannot perform arbitrary, unauthorized declassification, because the flow policies of all other principals are still maintained. Declassification permits applications running with sufficient authority (i.e., to act for a principal) to remove restrictions when appropriate; for example, an application might determine that the amount of information being leaked is inconsequential. Previous work on information flow did not allow any declassification within the model.

- IFlow supports efficient static analysis of information flow, which is required to prevent leaking information through implicit flows and to provide practical fine-grained control over information flow [DD77].

- Static checking is accomplished without imposing restrictions on the model that make it difficult or awkward to use. IFlow supports label polymorphism and safe run-time label checking. It supports label inference, which reduces the burden on programmers to add static information flow annotations to a program. Both label checking and label inference can be performed easily and efficiently.

- IFlow has a formal semantics that allows a precise characterization of which relabelings are legal. This definition lets us prove that the rules for static checking are both sound and complete: the rules allow only safe relabelings, and they allow all safe relabelings.

- IFlow allows a richer set of safe relabelings than do previous label models [Den76, MMN90], which do not exploit information about relationships between different principals.

- IFlow can be applied in dual form to yield decentralized integrity policies.

PART VI: Conclusions and Recommendations

13 Results

13.1 Access Control

13.1.1 Current Results

We have developed a policy language and mechanism that provide security support for information sharing between organizations. For instance, such a mechanism could be used to support the secure sharing of files, or Web pages, between organizations cooperating on some project. We have built a file sharing demonstration to illustrate this method.

Our approach enables different organizations and users to set appropriate fine-grained access controls with minimal user intervention. The basis of the approach is to use an access control language that can express access rules in terms of the relationships and attributes of users and objects in a general way. This permits the construction of policies that are appropriate to a wide class of target objects. When such policies are used as defaults for newly created objects, they are more likely to capture the desired security restrictions, and hence require less user modification.

As the basis for forming access rules, we used a variation on the object oriented constraint language, OCL. The intent was to permit access control to be expressed in terms of natural classes of users, target objects, and supporting concepts. Ideally, we could have just used OCL without modification, in order to conform to some object oriented standard. However, the OCL syntax is awkward in some places (such as expressing the containment of an element in a set), so we found it desirable to add some extensions to the language. One language feature of OCL that we kept, which perhaps we should have altered, is the use of the dot notation to indicate navigation both to attributes and to relations. In our access control language, they are conceptually quite different, and it might have been clearer if we had used two different syntactic constructs.

Because of the complexity of the OCL-like expression language, we added a simple front-end language based on access control lists. This may make it easier for end-users to make small adjustments to a policy. In other specific information sharing applications, there may be other simple front-end languages, or interfaces, that would also be useful.

Our system is designed to support sharing information in a way that permits organizations/users to allocate some of their resources to another user. The principal example is a directory owner who allows another user to create a subdirectory within his directory. The second user has control of the subdirectory, but the directory owner can still impose some access constraints on the subdirectory. To permit this scenario, we provide a way for a party to impose local policies on a directory (which only apply to that directory) and inheritable policies that also apply to all target objects in the directory.

Using this mechanism, multiple parties with a controlling interest in a given target can simultaneously impose access control policies on that target.

The specific classes that should be utilized in the specification and evaluation of the security rules depend on the needs of the participants of the information sharing system. For example, the best notion of a security integrity level may depend on the kind of data that needs to be protected. Hence, our system has been designed so that when it is installed, it can easily be configured to handle the appropriate classes.

13.1.2 Future Directions

We see several further directions for research. It would be interesting to perform field trials of our access control methodology to gauge the usability of the system. We may find that making some further syntactic modifications would enhance usability. We may also find that further front-end languages or interfaces might be useful.

Another direction would be to investigate more flexible mechanisms for management of meta-policy, such as modification rights on policies.

We could generalize the policy space from a tree hierarchy (policies on files and directories) to a space with more complex links than containment. Then the notion of policy inheritance might better reflect other useful notions of multiple controlling authorities.

13.2 Java Tools – PolyJ

13.2.1 Current Results

PolyJ is an extension of the Java programming language that provides notations for parameterized types. The PolyJ compiler can detect errors that would cause ordinary Java programs to fail at run-time. PolyJ also simplifies programming by eliminating the need for many explicit typecasts.

The PolyJ compiler was used to teach an undergraduate course in software engineering at MIT to over 100 students in both the spring and fall terms of 1998. The PolyJ compiler is freely available over the Internet at http://www.pmg.lcs.mit.edu/polyj. Development of PolyJ was supported jointly by this contract and by instructional funds from MIT.

Documentation for PolyJ includes a paper about its design [MBL97], Chapters 6 and 10 in a revised edition of *Abstraction and Specification* by Barbara Liskov and John Guttag [LG97], and a manual and `man` page included in the PolyJ distribution.

13.2.2 Future Directions

Sun Microsystems has submitted a Java Specification Request, JSR-000014, that proposes extending the Java programming language with generic types. This JSR lists

PolyJ as one of three competing proposals for this extension, the other two being GJ [BOSW98] and Nextgen [CS98].

PolyJ meets all of the constraints and goals in the JSR. In addition, it supports the use of primitive types as type arguments, a feature that the JSR describes as "nice," but "not ... a goal of the design." We think it should be a goal of the design, so that Java programmers will no longer have to cast an `int` to an `Int` in Java to use it as an element of a `Vector`. Furthermore, PolyJ's use of **where** clauses instead of interfaces as a means of constraining type arguments provides better support than GJ for program development. Programmers wanting to use a parameterized type may have access only to the object code, and not to the source code, for classes providing the arguments of that type. This presents no problem in PolyJ. However, GJ requires such access unless the classes used for arguments were written in a way that anticipated this use (i.e., they were specified as implementing the appropriate interfaces).

Because we believe PolyJ to be superior to GJ, the most important future direction for work on PolyJ is to try to influence the Java standardization effort to adopt its approach rather than GJ's. Unfortunately, GJ appears to have the inside track since it is being proposed by Sun itself.

13.3 Java Tools – Enhanced javadoc Utility

13.3.1 Current Results

We have developed a prototype, `percolator`, as an improved javadoc utility. Like `javadoc`, `percolator` extracts documentation from stylized comments embedded in source code. Unlike `javadoc`, `percolator` checks the extracted documentation for consistency with the source code. Hence, its use leads to documentation that is more accurate.

13.3.2 Future Directions

A tool based on `percolator` would be a useful addition to the collection of tools currently available for developing Java programs. One particularly useful enhancement to `percolator` would be support for PolyJ documentation, i.e., for checking that the documentation of parameterized types is consistent with the source code.

13.4 Enforcing Safety Properties

13.4.1 Current Results

With Naccio, it is possible to define a large and useful class of safety policies in a general and platform-independent way, and to enforce those policies on executions without an unreasonable performance penalty.

Naccio defines a safety policy by associating checking code with abstract resource manipulations. The policy definition mechanisms are general enough to describe a large

class of safety policies that comprises most useful policies. A standard resource library enables the creation of policies that are portable across Naccio implementations for different platforms. Naccio's policy definition mechanisms have considerable advantages over other alternatives. By describing policies in terms of abstract resource manipulations, they isolate policy authors from platform details. It is not necessary to know a particular platform API to produce or understand a safety policy. Once a policy has been developed, it can be reused on all platforms for which Naccio implementations are available.

Naccio's architecture for enforcing policies is based on transforming programs to insert checking code. This architecture replaces resource-manipulating calls with wrappers that perform checks before and after those calls. Low-level code safety mechanisms prevent program code from tampering with or circumventing the checking code. The enforcement architecture has two advantages over common alternatives. Because it modifies platform library object code directly, it does not require the availability of source code. Second, since it analyzes the policy statically and only introduces wrappers that are necessary for checking, the overhead required to enforce a policy is directly related to the amount of checking it does. If a policy does not constrain a particular resource manipulation, there is no checking overhead associated with that resource manipulation. The main drawback to the enforcement architecture is that it depends on a large trusted computing base. This increases the likelihood that there are vulnerabilities that can be exploited and makes assurance difficult.

We have developed two prototype Naccio implementations that enforce policies on JavaVM classes and Win32 executables. Naccio/JavaVM is a complete implementation. Naccio/Win32 does not yet provide a complete platform interface or implement the protective transformations necessary for low-level code safety. Although the prototype implementations are not ready for industrial deployment, they provide a proof-of-concept for the Naccio architecture.

Naccio represents one point in the design space for code safety systems. It is well suited to typical Internet users at small and medium size companies today and for the near future. It supports enforcement of a large class of policies with low preparation costs and with run-time overhead that is minimal for simple policies and that scales with the complexity of the policy. The current design is not well suited to high-security environments because its large trusted computing base makes assurance difficult.

13.4.2 Future Directions

For a code safety system to be trustworthy, there must be some assurance that it provides the expected security. As discussed in the previous section, one of the security vulnerabilities of Naccio is that it depends on a large trusted computing base. An industrial implementation should attempt to reduce the size of the trusted computing base and validate its most critical parts.

The prototype implementations are designed with ease of implementation as a priority. Although performance results indicate that the prototypes perform acceptably in most situations, industrial implementations could make substantial performance improvements. [Evans99b] discusses some straightforward ways to improve the performance of the policy compiler, program transformer, and executing application.

The prototypes do not include any tools to help policy authors write, understand and test policies. A better environment for developing policies is essential if policy authoring is to be accessible to non-experts. It would be useful to have tools that can automatically analyze policies and answer questions about what one policy allows that a different policy does not, or whether a policy always disallows a certain sequence of system calls. Suitable test cases can help detect simple errors in a policy, but they do not provide sufficient assurance that the policy means what its author intends.

Although the focus of our work on Naccio has been on code safety, Naccio has a number of other possible applications. Its mechanisms provide a way to alter or monitor the behavior of executions in ways that could be useful in addressing many other problems.

Naccio can be useful in *debugging* programs. For example, policies could be used to confirm that the number of bytes sent over the network is a function of the number of bytes read from files, or that every open file is closed before execution terminates, or that all files created in temporary directories are deleted. Policies used for debugging can be more precise than safety policies enforced on arbitrary programs since the programmers know a great deal about the expected behavior of their programs. For example, policy violations can be used to direct the programmer to examine assumptions about the behavior of the code more carefully, even when that code is behaving correctly.

Naccio policies can record program activity in logs file, which can be used for performance profiling and program analysis. Logging done at the system level would be useful for intrusion detection.

By altering platform interfaces, it is possible to change program behavior in ways that are not necessarily security related. For example, a policy can modify the behavior of a program to delay and split network transmissions to conform to a specified bandwidth constraint. Another policy could save backup versions of all files before they are overwritten.

By providing better ways to define safety policies along with efficient and convenient mechanisms for enforcing policies, we hope the situations in which code safety policies are used will be expanded. Currently, code safety is usually considered only for untrusted mobile code. A satisfactory code safety system would be useful in protecting users from bugs in applications from trustworthy sources as well. As the precision of safety policies increases and the costs of enforcement are reduced, policies can be enforced in more situations with more pervasive benefits.

13.5 Information Flow

13.5.1 Current Work

We have developed a decentralized model for information flow control, which can be used to address privacy and secrecy concerns in mobile code (such as downloaded Java applets). The model allows individual principals to attach flow policies to their data, to declassify data, and to act on behalf of other principals. The model enables efficient static checking of actual program code, which ensures that all principals' flow policies are respected (without leaking information, as do many run-time checks). [ML97, ML98].

A subsequent DARPA Contract (F30602-98-1-0237) has supported the development of JFlow, an extension of the Java programming language, together with a compiler for JFlow. Together, the language and the compiler provide an implementation of the IFlow model and methods developed under this contract. [Meyers99a, Meyers99b]

13.5.2 Future Directions

There are several directions in which to extend this work on information flow. One obviously important direction is to continue to make IFlow a more practical model for developing applications. IFlow addresses many of the limitations of earlier information flow systems that have prevented their use for the development of reasonable applications; however, more experience (e.g., with implementations such as JFlow) is needed to better understand the practical applications of this approach.

Our work has assumed an entirely trusted execution environment. The IFlow model does not work well in large, networked systems in which different principals may have different levels of trust in the various hosts in the network. One simple technique for dealing with distrusted nodes is to transmit opaque receipts or tokens for the data. Another approach is for a third party to provide a trusted host to get around the impasse of mutually distrusted hosts. It would be interesting to investigate a distributed computational environment in which secure computation is made transparent through the automatic application of these techniques.

Our work shows how to control several kinds of information flow channels better, including channels through storage, implicit flows, and run-time security checks. However, it does not treat covert channels that arise from timing channels and from the timing of asynchronous communication between threads. Supporting multi-threaded applications would make our work more widely applicable. Although there has been some work on analyzing these channels through static analysis, current techniques are overly restrictive. One central difficulty is the need to distinguish between locally and globally visible operations within a multi-threaded program. Current multi-threaded programming environments have tended to minimize this distinction, but without it, static analysis will not be a reasonably precise tool for controlling information flow. An altered

programming model may be possible in which enough information is available about inter-thread communication to permit precise analysis.

14 Technology Transfer Recommendations

- Advocate the PolyJ approach to polymorphism in the JSR-000014 Java standardization effort.
- Install and demonstrate Naccio at Air Force Research Lab - Rome site.
- Incorporate techniques used in the access control method into other policy related efforts.

References

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy, "Decentralized Trust Management", *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 164-173, 1996.

[BG98] Mark Bickford and David Guaspari, *Lightweight Analysis of UML*, ORA Tech Report TM98-98-0036, November 1998.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler, "Making the future safe for the past: adding genericity to the Java programming language", *Proceedings of the 13th ACM Conference on Object Oriented Programming, Systems and Applications (OOPSLA 98),* Vancouver, B.C., October 1998.

[CS98] Robert Cartwright and Guy Steele, "Compatible Genericity with Run-time Types for the Java(tm) Programming Language", *Proceedings of the 13th ACM Conference on Object Oriented Programming, Systems and Applications (OOPSLA 98),* Vancouver, B.C., October 1998.

[DD77] Dorothy E. Denning and Peter J. Denning, "Certification of programs for secure information flow", *Communications of the ACM*, 20(7), pp. 504-513, 1977.

[DGLM95] M. Day, R. Gruber, B. Liskov, and A. C. Myers, "Subtypes vs. `where` clauses: Constraining parametric polymorphism", *OOPSLA '95 Conference Proceedings*, ACM Press, pp. 156-158, October 1995.

[Den76] Dorothy E. Denning, "A lattice model of secure information flow", *Communications of the ACM*, 19(5), pp. 236-243, 1976.

[Eng96] David Engberg, *guavac, a free compiler for the Java language*, Effective Edge Solutions, San Francisco, 1996. ftp://ftp.yggdrasil.com/pub/dist/devel/compilers/guavac.

[Evans96] David Evans, *LCLint User's Guide, Version 2.2*, MIT Laboratory for Computer Science, August 1996.

[Evans99] David Evans, *Policy-Directed Code Safety*, PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1999.

[ET99] David Evans and Andrew Twyman, "Flexible Policy-Directed Code Safety", *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, 9-12 May 1999.

[FS97] Martin Fowler with Kendall Scott, *UML Distilled – Applying the Standard Object Modeling Language*, 1997.

[FR99] Francis Fung and David Rosenthal, *Software User Manual for the Access Control Object Language Demonstration*, Odyssey Research Associates, ORA-TM-99-0009, 1999.

[GG+97] Stephen Garland, David Guaspari, et al., *Year 1 Report – Security Engineering for High Assurance Policy-Based Applications*, Odyssey Research Associates, MIT, ORA-TM-97-0040, 1997.

[GH+92] John V. Guttag and James J. Horning, editors, with Stephen J. Garland, Kevin D. Jones, Andres Modet, and Jeannette M. Wing, *Larch: Language and Tools for Formal Specification*, Springer-Verlag, 1992.

[LCD+94] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers, *Theta Reference Manual*, Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994.

[LG97] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*, draft second edition, 1997.

[LY96] T. Lindholm and F. Yellin, *The Java Virtual Machine*, Addison-Wesley, Englewood Cliffs, NJ, May 1996.

[Myers99a] Andrew C. Myers, "JFlow: Practical Static Information Flow Control", *Proceedings of the 26th ACM Symposium on Principles of Programming Language (POPL '99)*, San Antonio, Texas, January 1999.

[Myers99b] Andrew C. Myers, *Mostly-Static Decentralized Information Flow Control*, PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1999.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov, "Parameterized Types for Java", *ACM POPL'97*, Paris, France, January 1997.

[ML97] Andrew C. Myers and Barbara Liskov, "A decentralized model for information flow control", *Proceedings of the 16th ACM Symposium on Operating System Principles*, Saint-Malo, France, 5-8 October 1997.

[ML98] Andrew C. Myers and Barbara Liskov, "Complete, safe information flow with decentralized labels", *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, 3-6 May 1998.

[MMN90] Catherine J. McCollum, Judith R. Messing, and LouAnna Notargiacomo, "Beyond the pale of MAC and DAC—defining new forms of access control", *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 190-200, 1990.

[Nel91] Greg Nelson (editor), *Systems Programming with Modula-3,* Prentice-Hall, 1991.

[OCL] OMG Unified Modeling Language Specification (draft), chapter 6: Object Constraint Language, Version 1.3a1, January 1999.

[ORW97] Martin Odersky, Enno Runne, Philip Wadler, *Two Ways to Bake Your Pizza – Translating Parameterised Types into Java*, Technical report CIS-97-016, School of Computer and Information Science, University of South Australia, 1997.

[Rat97a] Rational Software, et al., *Object Constraint Language Specification*, version 1.1, September 1997.

[Rat97b] Rational Software, et al., *UML Notation Guide*, version 1.1, September 1997.

[Rat97c] Rational Software, et al., *UML Semantics*, version 1.1, September 1997.

[RW97a] Martin Roscheisen and Terry Winograd, "A Network-Centric Design for Relationship-based Security and Access Control", *Journal of Computer Security*, 1997, Stanford (from Web http://mjosa.standford.edu/rmr/JoSec.html).

[RW97b] Martin Roscheisen and Terry Winograd, *The FIRM Framework for Interoperable Rights Management*, Draft, Stanford, April 1997.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[Sto87] B. Stoustrup, *The C++ Programming Language*, Addison-Wesley, 1987.

[Sun95a] Sun Microsystems, *Java Language Specification*, version 1.0 beta edition, October 1995.

[Sun95b] Sun Microsystems, *The Java Virtual Machine Specification*, release 1.0 beta edition, August 1995.

[Twy99] Andrew Twyman, *Flexible Code Safety for Win32*, MEng Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1999.

[WF98] Dan S. Wallach and Edward W. Felten, "Understanding Java Stack Inspection", *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, 3-6 May 1998.

[ZBCS97] Mary Ellen Zurko, Travis Broughton, Greg Carpenter, and Rich Simon, *An Architecture for Distributed Authorization*, The Open Group Research Institute, Oct. 1997.

MIT Web pages

Polyj http://www.pmg.lcs.mit.edu/polyj/

Code safety http://naccio.lcs.mit.edu/

☆U.S. GOVERNMENT PRINTING OFFICE: 2002/710-038/10266

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.